Last Man Standing: Static, Decremental and Dynamic Resiliency via Controller Synthesis

Matteo Zavatteri
Dipartimento di Informatica, Università di Verona, Italy
matteo.zavatteri@univr.it

Luca Viganò
Department of Informatics, King's College London, UK
luca.vigano@kcl.ac.uk

Abstract

The *workflow satisfiability problem* is the problem of finding an assignment of users to tasks (i.e., a plan) so that all authorization constraints are satisfied. The *workflow resiliency problem* is a *dynamic* workflow satisfiability problem coping with the absence of users. If a workflow is resilient, it is of course satisfiable, but the vice versa does not hold. There are three levels of resiliency: in *static* resiliency, up to k users might be absent before the execution starts and never become available for that execution; in *decremental* resiliency, up to k users might be absent before or during execution and, again, they never become available for that execution; in *dynamic* resiliency, up to k users might be absent *before executing any task* and they may in general turn absent and available continuously, before or during the execution. Much work has been carried out to address static resiliency, little for decremental resiliency and, to the best of our knowledge, for dynamic resiliency no exact approach that returns a dynamic execution plan if and only if a workflow is resilient has been provided so far. In this paper, we tackle workflow resiliency via *extended game automata*. We provide three encodings (having polynomial-time complexity) from workflows to extended game automata to model each kind of resiliency as an instantaneous game and we use Uppaal-TIGA to synthesize a winning strategy (i.e., a controller) for such a game. If a controller exists, then the workflow is *resilient* (as the controller's strategy corresponds to a dynamic plan). If it doesn't, then the workflow is *breakable*. The approach that we propose is *correct* because it corresponds to a reachability problem for extended game automata (TCTL model checking). Moreover, we have developed Erre, the first tool for workflow resiliency that relies on a controller synthesis approach for the three kinds of resiliency. Thanks to Erre, our approach is thus also *fully-automated* from analysis to simulation.

1 Introduction

1.1 Context and Motivation

Workflow technology has emerged as one of the leading technologies for modeling, (re)designing and executing business processes in several different application domains such as industrial R&D, manufacturing, energy distribution, banking processes, critical infrastructures and healthcare [1, 2]. A workflow is the automation of a business process, in whole or part, during which documents,

information or tasks are passed from one participant to another for action, according to a set of procedural rules. The conceptual modeling of workflows underlying business processes has been receiving increasing attention over the last years and many technical aspects have been discussed, including flexibility, structured vs. unstructured modeling, change management, authorization models, temporal features, resource allocation and constraints (see, e.g., [1, 3, 4, 5, 6]).

In such contexts, attention must especially be devoted to the resources (users, machineries, computational devices, etc.) employed in such workflows. Regarding resource allocation, workflows can be divided into (i) workflows where the availability of users is controlled (no one is assumed to become absent) and (ii) workflows where the availability of users is not controlled (sooner or later a user may become absent). The first case, is the classic *workflow satisfiability problem* (*WSP*, [4, 5]) in which it is enough to come up with an assignment of tasks to users (i.e., a plan) satisfying the underlying constraint satisfaction problem [7]. In the second case, we must operate according to the kind of uncertainty we are dealing with to keep guaranteeing that the security policies we expect to hold will never be broken. For example, conditional uncertainty models workflows where we cannot decide which conditional path to take during execution, whereas resource uncertainty models workflows where the availability of resources is unknown during execution.

Specifically, to guarantee a successful task to user assignment under conditional uncertainty that always satisfies a given set of security policies in [8, 9] we build on [6] to provide *constraint networks under conditional uncertainty (CNCUs)* as an extension of classic *constraint networks (CNs, [7]).* CNCUs address natively conditional uncertainty by means of observation points in the workflow allowing us to observe the uncontrollable behavior (of the environment) revealing in real time which workflow path the execution will go through. Since the choice of the workflow path to take is out of control (and only revealed during execution) the same tasks may in general be assigned to different users depending on what is going on. In such a context, the classic workflow satisfiability *does fail* because the synthesized plan it relies on is fixed and unable to handle conditional assignments leading to eventually break a subset of the security policies we expect to hold.

Consider now an unconditional workflow in which the availability of users is out of control.

What can go wrong?

Some users might wake up feeling sick and call in to say that they are not going to come to work. Other users could be involved in a traffic jam on their way to work or simply decide to go on strike. In such a case, we must guarantee a successful execution with the only remaining users who agree not to leave until the workflow completes.

Are we safe now?

No, we are not. The situation could get worse than the previous one. Some of the users (with which the execution started) might get a call announcing an emergency in the family. In such a case, we must guarantee a successful execution with, again, the only remaining users who might, this time, leave before the workflow completes.

Are we safe now?

Still, no. We could have our worst day ever. Some users might not arrive in time at work or might not arrive at all. Others could leave before the workflow ends and never come back in time to finish the work they started. Others could be absent for a short period of time (e.g., to go to the doctor to collect some medical prescriptions). And so on. In such a case, we must guarantee that

before executing any task we have enough users left to keep executing the tasks until the workflow completes.

Now we are safe.

The previous three examples provide the intuitions behind the three main kinds of workflow resiliency defined by Wang and Li in [5] (initially in [4]):

- *Static resiliency* is when users are absent before starting and do not come back (as in the first scenario above).
- *Decremental resiliency* is when they can also become absent during execution but, again, they do not come back (second scenario).
- *Dynamic resiliency* is when (possibly different sets of) users can become and stay absent for any (possibly big) time interval; they can also come back and become absent over and over again (third scenario).

Some works have addressed workflow resiliency probabilistically, e.g., [10, 11], whereas other works addressed it by modifying the constraints, e.g., [12, 13]. Several approaches consider static resiliency only (e.g., [14, 15, 16]), one of them also considers decremental resiliency [16], whereas, to the best of our knowledge, an exact approach to dynamic resiliency (i.e., an approach that returns a dynamic execution plan if and only if a workflow is resilient without modifying the problem nor returning an execution plan that may fail) remains unexplored; see also [17] for a very recent survey on workflow satisfiability and resiliency.

1.2 Contributions

We address dynamic resiliency (and therefore also static and decremental resiliency) by moving our analysis from satisfiability to *controllability*. Our contributions in this paper are three-fold.

1. We provide three encodings into extended game automata to model static, decremental and dynamic resiliency (up to maximum k absent users) as two-player games, and we discuss how to get dynamic plans (or prove that none exists) for each kind of resiliency via controller synthesis by using Uppaal-TIGA. In each encoding the maximum number of absent users is given as input. For static resiliency we aim to synthesize a plan saying how to execute the workflow depending on which subset of users is absent before starting the execution. For decremental resiliency the plan would also tackle users that might turn absent during execution, whereas for dynamic resiliency the plan deals with users that (possibly) come and go before the execution of any task.
2. We prove: the correctness of the encodings, that these encodings are realizable in polynomial time and that dynamic resiliency is a matter of order.
3. We introduce Erre, a tool that we have developed to automate and carry out an initial experimental evaluation and we discuss how we generated the related set of benchmarks. Erre is the first tool for workflow resiliency that relies on a controller synthesis approach for the three kinds of resiliency.

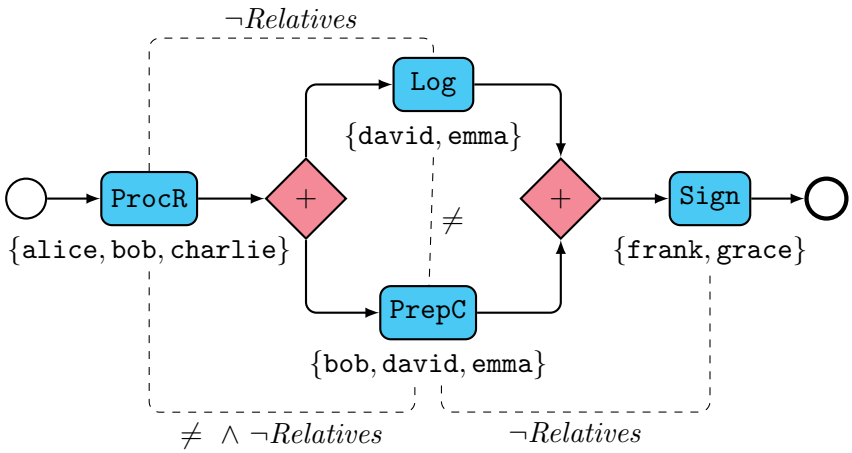


Figure 1: A simplification of a loan origination process. Solid edges model the partial order between tasks, whereas dashed ones model authorization constraints.

1.3 Organization

Section 2 introduces a motivating example that we use throughout the paper. Section 3 provides essential background on extended game automata as well as workflow satisfiability and resiliency. Section 4 provides the encodings from workflows into extended game automata for static, decremental and dynamic resiliency as well as the related controller synthesis phase to get dynamic plans. Section 5 discusses the correctness and complexity of the encodings provided in Section 4. Section 6 presents our tool Erre, along with an experimental evaluation (Erre is currently the only tool able to tackle all kinds of resiliency in an exact way). Section 7 discusses related work. Section 8 sums up and discusses future work.

2 A Motivating Example

As a motivating example, we consider a simplification of a *loan origination process (LOP)* for eligible customers whose financial records have already been approved. We show the workflow in Figure 1 and we recall that *separation of duty* requires that the users executing a subset of tasks are different, whereas *binding of duties* requires that the users executing a subset of tasks are equal. For our example all separations of duty involve two tasks only.

The workflow starts with a pre-processing clerk (`alice` or `bob` or `charlie`), who processes a loan request (`ProcR`). After that, the flow of execution splits unconditionally (leftmost diamond labeled by +) and enters a parallel block, where an auditor (`david` or `emma`), who must not be a relative of the user who executed `ProcR`, logs the just processed request for future accountability purposes (`Log`). Simultaneously, a post-processing clerk (`bob` or `david` or `emma`), who must be different from and not a relative of the user who executed `ProcR` and also different from the user who executed (or will execute) `Log`, prepares the contract (`PrepC`). The order of `Log` and `PrepC` does not matter. Finally, the flow of execution joins and exits the parallel block (rightmost diamond labeled by +) where a manager (`frank` or `grace`), who must not be a relative of the user who executed `PrepC`, signs the contract (`Sign`). Note that some users may belong to more than one role (e.g., `bob` is both a pre-processing and a post-processing clerk). In Figure 1 directed edges model the partial

order among tasks, whereas undirected dashed edges model authorization constraints. We show authorized users below tasks. For the sake of the example, `bob` and `david` are brothers, whereas `emma` is `frank`'s daughter.

Our goal is to always execute this workflow satisfying all authorization constraints in particular when some users are or will potentially become absent during execution.

3 Background

In this section, we first summarize workflow satisfiability and resiliency and then we summarize deterministic finite automata (DFAs) [18], game automata (GAs) and their extensions to support (bounded) integer variables: extended game automata (EGAs). EGAs are the fragment of extended timed game automata [19] in which data variables are restricted to be of integer type and time aspects are neglected (see, for example, the discussion in [20, 21] for a definition of extended timed game automata).

3.1 Workflow Satisfiability and Resiliency

The core of a *workflow* defines a set of tasks and a partial-order relation saying in which order such tasks have to be executed. An *access-controlled workflow* extends a classic workflow by adding a set of users, an authorization relation and a set of constraints saying which combinations of task assignments to users are permitted [4, 5]. In this paper we give a specification of access controlled workflow whose set of constraints consists of entailment and counting constraints [22]. More formally,

Definition 1. *An* access-controlled workflow (ACWF) *is a tuple* $W = \langle T, U, \preceq, A, C \rangle$*, where:*

1. $T = \{t_1, \ldots, t_n\}$ *is a* finite *set of tasks (atomic work units).*
2. $U = \{u_1, \ldots, u_m\}$ *is a* finite *set of users (the resources to commit for executing tasks).*
3. $\preceq \subseteq T \times T$ *is a* partial-order relation*. If* $(t_1, t_2) \in \preceq$ *(or simply* $t_1 \preceq t_2$*), then* t_1 *is executed* before t_2*. Tasks* t_1 *and* t_2 *may be executed simultaneously if neither* $t_1 \preceq t_2$ *nor* $t_2 \preceq t_1$ *are specified.*
4. $A \subseteq T \times U$ *is the* authorization relation*. We abuse notation and write* $A(t) = \{u \mid (t, u) \in A\}$ *to shorten the set of users authorized for* t*.*
5. C *is a set of* entailment and counting constraints*. An* entailment constraint *has the form* (ρ, T_1, T_2)*, where* $\rho \subseteq U \times U$ *and* $T_1, T_2 \subseteq T$ *and it is a binary relational constraint between the authorized users of two tasks. A* counting constraint *has the form* (x, y, T_1)*, where* $x, y \in \mathbb{N}$*,* $0 \leq x \leq y \leq |T_1|$ *and* $T_1 \subseteq T$ *and it is a cardinality constraint imposing a minimum* (x) *and a maximum* (y) *number of tasks a user* $u \in U$ *can be assigned if* u *executes some* $t \in T_1$*.* □

Entailment constraints are basically a kind of relational constraint and divide in three main types:

Type 1: when $|T_1| = |T_2| = 1$

Type 2: when $|T_1| = 1$ and $|T_2| > 1$

Type 3: when $|T_1| > 1$ and $|T_2| > 1$

Counting constraints are basically cardinality constraints and are sometimes a compact way to encode certain entailment constraints.

Example 1. Let $\neq$ be defined as $\{(u_1, u_2) \mid u_1, u_2 \in U \wedge u_1 \neq u_2\}$ and $=$ as $\{(u_1, u_2) \mid u_1, u_2 \in U \wedge u_1 = u_2\}$, then the entailment constraints $(\neq, \{t_1\}, \{t_2\})$ and $(=, \{t_1\}, \{t_2\})$ are equivalent to the counting constraints $(1, 1, \{t_1, t_2\})$ and $(2, 2, \{t_1, t_2\})$ modeling a separation and a binding of duties, respectively (to understand why see the satisfaction of constraints below). More concretely, consider Figure 1 and its formal specification in Example 3. We model the dashed edge between `ProcR` and `PrepC` labeled by "$\neq \wedge \neg$*Relatives*" with a counting (left) plus an entailment constraint (right) as follows:

$$\neq \qquad \wedge \qquad \neg \mathit{Relatives}$$

becomes

$$\underbrace{(1, 1, \{\texttt{ProcR}, \texttt{PrepC}\})}_{\text{Counting constraint}} \qquad \mathit{plus} \qquad \underbrace{(\neg \mathit{Relatives}, \{\texttt{ProcR}\}, \{\texttt{PrepC}\})}_{\text{Entailment constraint}}$$

□

A *plan* says how a workflow is executed, i.e., which tasks are assigned to which users. Formally:

Definition 2. *A* plan *is a mapping $\pi\colon T \to U$ assigning tasks to users. A plan π*

- *is* authorized *for a workflow $\langle T, U, \preceq, A, C\rangle$ if $\pi(t) \in A(t)$ (i.e., $(t, \pi(t)) \in A$) for every $t \in T$;*
- satisfies an entailment constraint (ρ, T_1, T_2) *if there exist $t_1 \in T_1$ and $t_2 \in T_2$ such that $(\pi(t_1), \pi(t_2)) \in \rho$;*
- satisfies a counting constraint (x, y, T_1) *if any user who executes some tasks in T_1, executes from x to y (different) tasks in that set (the constraint is trivially satisfied for all users not executing any task in T_1);*
- *is* consistent *if it is authorized and satisfies all constraints.* □

Example 2. Continuing Example 1, the constraint $(1, 1, \{t_1, t_2\})$ says that if a user executes some $t \in \{t_1, t_2\}$, then he must execute exactly 1 task (separation of duty), whereas the constraint $(2, 2, \{t_1, t_2\})$ says that if a user executes some $t \in \{t_1, t_2\}$, then he must execute exactly 2 tasks (all) in $\{t_1, t_2\}$ (binding of duties). Once again, getting back to Figure 1 and Example 2 the counting constraint $(1, 1, \{\texttt{ProcR}, \texttt{PrepC}\})$ says that any user authorized for `ProcR` and `PrepC` does not execute both tasks. Consider `alice`, `charlie`, `david` and `emma`. Each of these users is authorized either for `ProcR` or for `PrepC`, therefore, each user can execute at most one task. Instead, `bob` is authorized for both. Therefore, either `bob` doesn't execute `ProcR` nor `PrepC`, or `bob` must execute only one of them (no matter which one). □

Definition 3. *An ACWF is* satisfiable *if there exists a total ordering on the tasks meeting the restrictions imposed by $\preceq$ (topological sort) and a consistent plan for it.* □

Example 3. The formal specification of the ACWF in Figure 1 is

1. $T = \{\texttt{ProcR}, \texttt{Log}, \texttt{PrepC}, \texttt{Sign}\}$.
2. $U = \{\texttt{alice}, \texttt{bob}, \texttt{charlie}, \texttt{david}, \texttt{emma}, \texttt{frank}, \texttt{grace}\}$.
3. $\texttt{ProcR} \preceq \texttt{Log}, \texttt{ProcR} \preceq \texttt{PrepC}, \texttt{Log} \preceq \texttt{Sign}, \texttt{PrepC} \preceq \texttt{Sign}$.
4. $A(\texttt{ProcR}) = \{\texttt{alice}, \texttt{bob}, \texttt{charlie}\}$, $A(\texttt{Log}) = \{\texttt{david}, \texttt{emma}\}$, $A(\texttt{PrepC}) = \{\texttt{bob}, \texttt{david}, \texttt{emma}\}$ and $A(\texttt{Sign}) = \{\texttt{frank}, \texttt{grace}\}$.[1]
5. $C = \{(\neg \mathit{Relatives}, \{\texttt{ProcR}\}, \{\texttt{Log}\}), (\neg \mathit{Relatives}, \{\texttt{ProcR}\}, \{\texttt{PrepC}\}), (\neg \mathit{Relatives}, \{\texttt{PrepC}\}, \{\texttt{Sign}\}), (1, 1, \{\texttt{ProcR}, \texttt{PrepC}\}), (1, 1, \{\texttt{Log}, \texttt{PrepC}\})\}$.

where $\neg \mathit{Relatives}$ is denoted as $U \times U \setminus \{(\texttt{bob}, \texttt{david}), (\texttt{david}, \texttt{bob}), (\texttt{emma}, \texttt{frank}), (\texttt{frank}, \texttt{emma})\}$, whereas the counting constraints model (in a compact form) the various separation of duty discussed in Section 2. The first three entailment constraints in C refer to the dashed edges between ProcR and Log (whole label), ProcR and PrepC (second part of the label) and PrepC and Sign (whole label) in Figure 1, respectively. The last two refer to the dashed edges between ProcR and PrepC (first part of the label) and Log and PrepC (whole label) in Figure 1, respectively.

The workflow in Figure 1 is satisfiable. A possible total ordering is

$$\texttt{ProcR} \preceq \texttt{Log} \preceq \texttt{PrepC} \preceq \texttt{Sign}$$

and a consistent plan is

$$\pi(\texttt{ProcR}) = \texttt{alice},\ \pi(\texttt{Log}) = \texttt{david},\ \pi(\texttt{PrepC}) = \texttt{bob},\ \pi(\texttt{Sign}) = \texttt{frank},$$

meaning that alice processes the request, david logs it, bob prepares the contract and frank signs it. □

Workflow satisfiability assumes that all users are *always* available (that's why a static plan is enough). However, when the availability of users is uncertain, workflow satisfiability is not enough to guarantee that a consistent plan will never break any security policy. In such a case, we need to understand if the workflow is *resilient*. Indeed, *workflow resiliency* calls for the synthesis of *dynamic plans* whose assignments of tasks to users are done dynamically *while* the workflow is being executed according to which users are *available* and which ones are *absent*. More technically, let U be the set of users, $|U|$ the number of users in U and k the maximum number of absent users, where $k \in \mathbb{N}$ and $0 \leq k < |U|$. If $k = 0$, then no user is (or will become) absent. Otherwise, up to k-users are (or will become) absent before or during execution. These users can also remain absent or come back according to the type of resiliency. The analysis obviously does not make sense for $k = |U|$: the workflow is trivially not resilient because the strategy of the environment is "make all users absent and don't do anything else" (that's why this case is excluded). The analysis makes sense when $k < |U|$, where the special case $k = 0$ boils down to WSP. In dynamic resiliency, different subsets of users might be absent at different points of the execution. k indicates that any subset $U' \subset U$ of absent users (no matter which one and at which point of the execution) has cardinality $0 \leq |U'| \leq k < |U|$. As a result, $U \setminus U'$ represents the set of available users (however, this does not imply that all $A(T) \neq \emptyset$ once k users are made absent).

In [4, 5], Wang and Li defined three levels of resiliency:

[1] More formally, $A = \{(\texttt{ProcR}, \texttt{alice}), (\texttt{ProcR}, \texttt{bob}), (\texttt{ProcR}, \texttt{charlie}), (\texttt{Log}, \texttt{david}), (\texttt{Log}, \texttt{emma}), (\texttt{PrepC}, \texttt{bob}), (\texttt{PrepC}, \texttt{david}), (\texttt{PrepC}, \texttt{emma}), (\texttt{Sign}, \texttt{frank}), (\texttt{Sign}, \texttt{grace})\}$.

Algorithm 1: StaticResiliency

Input: $W = \langle T, U, \preceq, A, C \rangle$ and $0 \leq k < |U|$
Output: Resilient if W is statically resilient up to k absent users. Breakable otherwise.

The environment chooses $\texttt{Absent} \subset U$ such that $|\texttt{Absent}| \leq k$
$U \leftarrow U \setminus \texttt{Absent}$
if *there exists a total ordering on tasks and a consistent plan* π *for* W **then**
  return *Resilient* ▷ controller wins
return *Breakable* ▷ environment wins

Algorithm 2: DecrementalResiliency

Input: $W = \langle T, U, \preceq, A, C \rangle$ and $0 \leq k < |U|$
Output: Resilient if W is decrementally resilient up to k absent users. Breakable otherwise.

$\texttt{Absent} \leftarrow \emptyset$
$\pi \leftarrow$ initial plan
$\texttt{Executed} \leftarrow \emptyset$
while $\texttt{Executed} \neq T$ **do**
  The environment chooses $U' \subset U$ such that $|U' \cup \texttt{Absent}| \leq k$
  $\texttt{Absent} \leftarrow \texttt{Absent} \cup U'$
  $U \leftarrow U \setminus \texttt{Absent}$
  The controller looks for $t \in T \setminus \texttt{Executed}$ (coherent with $\preceq$ in the current execution) and $u \in U$ such that $(t, u) \in A$ and $\pi \cup \{\pi(t) = u\}$ belongs to a possible consistent plan
  if *no pair* (t, u) *exists* **then**
    return *Breakable* ▷ environment wins
  $\pi \leftarrow \pi \cup \{\pi(t) = u\}$
  $\texttt{Executed} \leftarrow \texttt{Executed} \cup \{t\}$
return *Resilient* ▷ controller wins

1. *Static resiliency*: up to $k < |U|$ users might be absent *before* the execution of the workflow starts and these users will never become available for that execution.

2. *Decremental resiliency*: up to $k < |U|$ users might be absent *before* execution or become so during it. As in the static case, absent users will never become available again for that execution.

3. *Dynamic resiliency*: up to $k < |U|$ (possibly different) users might be absent *before executing any task*. These users may become absent and become available again (possibly) many times (i.e., they come and go).

As is evident (and shown more formally in [4, 5]), dynamic resiliency entails decremental resiliency, which entails static resiliency.

Each of these levels of resiliency can be seen as a two-player game where a controller dynamically generates an ordering and a plan π (such that the ordering will be total and coherent with $\preceq$ and the plan consistent at the end of the execution), while the environment makes users absent in order to break consistency of π. If the controller wins the game, then the workflow is resilient, whereas if it loses, then the workflow is breakable as the environment has a strategy to *break* any potentially consistent π. Of course, when the number of absent users is $k = 0$, all three levels of resiliency boil down to classic workflow satisfiability (as all users are available).

Algorithm 3: DynamicResiliency

Input: $W = \langle T, U, \preceq, A, C \rangle$ and $0 \leq k < |U|$
Output: Resilient if W is dynamically resilient up to k absent users. Breakable otherwise.

$\pi \leftarrow$ initial plan
$\texttt{Executed} \leftarrow \emptyset$
while $\texttt{Executed} \neq T$ **do**
  The environment chooses $\texttt{Absent} \subset U$ such that $|\texttt{Absent}| \leq k$
  $U' \leftarrow U \setminus \texttt{Absent}$
  The controller looks for $t \in T \setminus \texttt{Executed}$ (coherent with $\preceq$ in the current execution) and $u \in U'$ such that $(t, u) \in A$ and $\pi \cup \{\pi(t) = u\}$ belongs to a possible consistent plan
  if *no pair* (t, u) *exists* **then**
    return *Breakable* ▷ environment wins
  $\pi \leftarrow \pi \cup \{\pi(t) = u\}$
  $\texttt{Executed} \leftarrow \texttt{Executed} \cup \{t\}$
return *Resilient* ▷ controller wins

Algorithm 1, Algorithm 2 and Algorithm 3 summarize the formalizations of the games proposed by Wang and Li in [4, 5] to model static, decremental and dynamic resiliency.

Wang and Li also proved (with respect to their specification) that deciding workflow satisfiability is NP-hard, deciding static resiliency is in coNP^{NP}, whereas deciding decremental and dynamic resiliency is PSPACE-complete [5]. Recently, Fong proved that deciding static resiliency is coNP^{NP}-complete [23].

The workflow in Figure 1 is statically, decrementally and dynamically resilient up to $k = 1$ (see the discussion in Section 6).

3.2 Extended Game Automata

Deterministic finite automata (*DFAs*, [18]) are the core of many modern state-transition systems that have pervasively become part of our life such as light switches, gate remotes, washing machines, various hardware components in modern computers and many other computational devices. DFAs allow for the specification of an abstract system whose behavior is defined according to a finite set of possible states and a finite set of transitions regulating the evolution of the system (i.e., the changing of state) according to the provided permitted actions that were decided as design time. For instance, flicking a light switch may result in either turning on or off the light depending on the previous state of the system.

However, DFAs fail to model systems dealing with *uncontrollable* actions (i.e., actions carried out by the environment). To overcome this limitation, *game automata* (*GA*, [24]) extended classic DFAs by dividing the set of transitions into *controllable* and *uncontrollable*. Controllable transitions are assigned to a controller (us), whereas uncontrollable transitions are assigned to the environment. In any state of the GA, both players are not obliged to play, i.e., take one of their transitions (if any). Conversely, when there exists a state from which both of them decide to take a transition, the environment plays first as uncontrollable transitions have *priority* over controllable ones (this is inherited from the semantics of timed game automata [19]). Taking an uncontrollable transition results in ignoring the controllable one that the controller decided to take (if it meant to play in that state). The purpose of the controller is to *reach an accepting state*, whereas that of the

environment is to prevent the controller from doing so. An accepting state (also known as *final state*) is a state of the automaton that, if reached, implies that the sequence of transitions taken to go from the initial state to it forms a word belonging to the language accepted by the automaton.

Therefore, in order to get to one of the accepting states we need something more than a mere sequence of transitions, we need a *strategy*: a mapping from states to controllable transitions guaranteeing that, if followed, we will eventually end up in one of the accepting states.

However, GAs do not model games which also consider integer variables and operations on them that may appear in the guards and/or updates of transitions. To achieve this purpose, we consider a fragment of timed game automata extended with integer variables discussed in [21, 25, 20, 26]. We refer to this fragment as *extended game automata (EGAs)*. Recall that when adding integer variables to GAs and operations on them (such as sum, subtraction, multiplication, divisions, Boolean comparison etc.), decidability of model checking does not break provided that these variables are bounded (i.e., their domains are finite) [21, 25, 20, 26].

In [21, 25, 20, 26] there is no clear consensus on the formal definition of EGAs, so, as a minor contribution, we provide in this paper a possible formal definition that contains just what we need. To that end, given a finite set of integer variables I in which each variable $\mathtt{i} \in I$ has a finite domain $\mathtt{dom(i)} = \{\underline{\mathtt{i}}_1, \underline{\mathtt{i}}_2, \dots\} \subset \mathbb{Z}$ (with $\underline{\mathtt{i}}_1$, $\underline{\mathtt{i}}_2$,... integer constants), let $\Gamma = \{\mathtt{a} \mapsto \underline{\mathtt{a}}, \mathtt{b} \mapsto \underline{\mathtt{b}}, \dots\}$ be a set of variable assignments to all integer variables in I where $\underline{\mathtt{a}} \in \mathtt{dom(a)}$, $\underline{\mathtt{b}} \in \mathtt{dom(b)}$ and let Γ^* be the set of all possible combination of value assignments. Note that since I is finite and any $\mathtt{i} \in I$ has a finite domain $\mathtt{dom(i)}$, we have that Γ^* is finite as well.

Definition 4. *An* extended game automaton (EGA) *is a tuple* $\langle L, \Sigma, I, \delta, q_0, \Gamma_0, F\rangle$*, where:*

1. $L = \{\ell_0, \ell_1, \dots\}$ *is a* finite *set of* locations*.*
2. $\Sigma = \{a, b, \dots\}$ *is a* finite *set of* input symbols *modeling the possible actions.*
3. $I = \{\mathtt{i}, \dots\}$ *is a finite set of integer variables each one with an associated bounded domain* $\mathtt{dom(i)}, \dots$*.*
4. $\delta \subseteq L \times G \times \Sigma \times V \times L$ *is a partial function representing the* transition relation*, where G is a Boolean expression involving conjunctions, disjunctions, negations and parenthesization and whose atoms have the form $\mathtt{i} \sim \underline{\mathtt{c}}$ where $\mathtt{i} \in I$, $\underline{\mathtt{c}}$ is a constant and $\sim \in \{<, >, =, \neq, \geq, \leq\}$, Σ is a set of input symbols and V is a (possibly empty) sequence of variable assignments $\mathtt{i} := \mathtt{i} * \underline{\mathtt{c}} + \underline{\mathtt{d}}$, where $\mathtt{i} \in I$ and $\underline{\mathtt{c}}, \underline{\mathtt{d}}$ are integer constants. For any $\langle \ell_i, G, a, \{\mathtt{i}_1 := \mathtt{i}_1 * \underline{\mathtt{c}}_1 + \underline{\mathtt{d}}_1, \mathtt{i}_2 := \mathtt{i}_2 * \underline{\mathtt{c}}_2 + \underline{\mathtt{d}}_2, \dots\}, \ell_j\rangle$, if the guard G is true, the EGA can move from ℓ_i to ℓ_j by evolving the current Γ into Γ' according to the assignments $\mathtt{i}_1 := \mathtt{i}_1 * \underline{\mathtt{c}}_1 + \underline{\mathtt{d}}_1, \mathtt{i}_2 := \mathtt{i}_2 * \underline{\mathtt{c}}_2 + \underline{\mathtt{d}}_2, \dots$ (considered in this order) realizing action a. Again, transitions are divided in controllable and uncontrollable with uncontrollable transitions having priority over controllable ones.*
5. $\ell_0 \in L$ *is the* initial *location.*
6. $\Gamma_0 = \{\mathtt{a} \mapsto \underline{\mathtt{a}}, \mathtt{b} \mapsto \underline{\mathtt{b}}, \dots\} \in \Gamma^*$ *represents the initialization of all integer variables in I.*
7. $F \subseteq L$ *is the set of* accepting *locations.*

A state *of an EGA is a pair (ℓ, Γ), where $\ell \in L$ and $\Gamma \in \Gamma^*$.* □

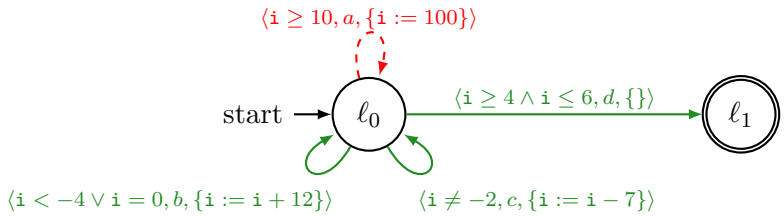


Figure 2: Controllable Extended Game Automata. Solid edges (green) model controllable transitions, whereas dashed ones (red) uncontrollable ones.

We graphically represent an EGA as a (multi)graph where the set of nodes coincides with L and the set of edges is such that there exists an edge $\ell_i \to \ell_j$ labeled by $\langle g, a, v\rangle$ for each $(\ell_i, g, a, v, \ell_j) \in \delta$.

A (memoryless) execution strategy (or positional strategy) for an EGA is a partial function $\sigma\colon L \times \Gamma^* \to \delta$ telling the controller which transition to take depending on the current state (ℓ, Γ). Once again, for the purpose of this paper, an EGA is controllable if there exists a winning strategy that guarantees the controller to always reach an accepting location.

Example 4. Figure 2 shows an example of EGA with two locations $L = \{\ell_0, \ell_1\}$ (with ℓ_0 initial and $F = \{\ell_1\}$), one integer variable $I = \{\mathtt{i}\}$ having domain $\mathtt{dom(i)} = \{x \mid -100 \leq x \leq 100\}$ and such that $\Gamma_0 = \{\mathtt{i} \mapsto 0\}$, an uncontrollable transition τ_1 and three controllable ones τ_2, τ_3, τ_4 specified as follows:

τ_1: $\langle \ell_0, \mathtt{i} \geq 10, a, \{\mathtt{i} := 100\}, \ell_0\rangle$ meaning if the current location is ℓ_0 and $\mathtt{i} \geq 10$, then realize action a incrementing $\mathtt{i}$ by 100 and remaining in ℓ_0.

τ_2: $\langle \ell_0, \mathtt{i} < -4 \vee \mathtt{i} = 0, b, \{\mathtt{i} := \mathtt{i} + 12\}, \ell_0\rangle$ meaning if the current location is ℓ_0 and $\mathtt{i} < 10$ or $\mathtt{i} = 0$, then realize action b incrementing $\mathtt{i}$ by 12 and remaining in ℓ_0.

τ_3: $\langle \ell_0, \mathtt{i} \neq -2, c, \{\mathtt{i} := \mathtt{i} - 7\}, \ell_0\rangle$ meaning if the current location is ℓ_0 and $\mathtt{i} \neq -2$, then realize action c decrementing $\mathtt{i}$ by 7 and remaining in ℓ_0.

τ_4: $\langle \ell_0, \mathtt{i} \geq 4 \wedge \mathtt{i} \leq 6, d, \{\}, \ell_1\rangle$ meaning if the current location is ℓ_0 and $\mathtt{i} \geq 4$ and $\mathtt{i} \leq 6$, then realize action d without updating any integer variable and enter ℓ_1.

The EGA is controllable. To prove that consider the following strategy

$$\sigma(\ell_0, \{\mathtt{i} \mapsto 0\}) = \tau_3 \qquad \sigma(\ell_0, \{\mathtt{i} \mapsto -7\}) = \tau_2 \qquad \sigma(\ell_0, \{\mathtt{i} \mapsto 5\}) = \tau_4$$

This strategy says that in $(\ell_0, \{\mathtt{i} \mapsto 0\})$ the controller must take τ_3 to move to $(\ell_0, \{i \mapsto -7\})$ (i.e., to remain in ℓ_0 decrementing $\mathtt{i}$ by 7). Then, in $(\ell_0, \{\mathtt{i} \mapsto -7\})$ the strategy says to take τ_2 to move to $(\ell_0, \{\mathtt{i} \mapsto 5\})$ (i.e., to remain in ℓ_0 incrementing $\mathtt{i}$ by 12). Finally, in $(\ell_0, \{\mathtt{i} \mapsto 5\})$ it says to take τ_4 to move to $(\ell_1, \{\mathtt{i} \mapsto 5\})$ and win the game.

Yet, one could wonder if in $(\ell_0, \{i \mapsto 0\}$ we could have taken τ_2 to get to $(\ell_0, \{i \mapsto 12\})$, then τ_3 to get to $(\ell_0, \{i \mapsto 5\})$ and finally τ_4 to move to $(\ell_1, \{i \mapsto 5\})$. That is, one could wonder if the strategy

$$\sigma'(\ell_0, \{\mathtt{i} \mapsto 0\}) = \tau_2 \qquad \sigma'(\ell_0, \{\mathtt{i} \mapsto 12\}) = \tau_3 \qquad \sigma'(\ell_0, \{\mathtt{i} \mapsto 5\}) = \tau_4$$

would have been another winning strategy. The answer is no and the reason is that σ' does not guarantee to reach q_1 because in $(\ell_0, \{\mathtt{i} \mapsto 12\})$ the environment can take τ_1 which, having priority,

moves to $(\ell_0, \{\mathtt{i} \mapsto 100\})$ from which no controllable transition can be taken to get to ℓ_1. Likewise, the EGA in Figure 2 turns uncontrollable if τ_1 is modified into $\tau_1' : \langle \ell_0, \mathtt{i} \geq 0, a, \{\mathtt{i} := 100\}, \ell_0 \rangle$ since the environment can take τ_1' in $(\ell_0, \{\mathtt{i} \mapsto 0\})$ moving to $(\ell_0, \{\mathtt{i} \mapsto 100\})$ because, again, uncontrollable transitions go first. □

In this paper, we are only interested in this kind of (pure) *reachability games* (i.e., getting to some accepting state) and we rely on UPPAAL-TIGA [25] as an off the shelf model checker for extended (timed) game automata in order to synthesize the strategies we need, or to prove that none exists by synthesizing the environment's. In UPPAAL-TIGA, we can specify queries in *timed computation tree logic (TCTL)*, an extension of CTL [27] supporting time which was proposed in [28]. Although we don't need time aspects in this paper, we rely on this software anyway as untimed games are a particular case of timed games in which time aspects can be neglected and also because UPPAAL-TIGA is a state of the art tool. Therefore, when synthesizing strategies (i.e., controllers) for such two-player games we will be always model checking our EGAs against TCTL formulae having the simple form

$$A\Diamond(q_i \vee \cdots \vee q_j)\,,$$

where $\{\ell_i, \ldots, \ell_j\} = F$. That is, in all paths (i.e., possible different runs of the game) the controller eventually reaches an accepting state (ℓ, Γ) such that $\ell \in F$ (i.e., it always eventually wins the game).

4 Workflow Resiliency via EGAs

In this section, we address all three kinds of resiliency. We provide three encodings from ACWFs into EGAs to check static, decremental and dynamic resiliency and we start by discussing the core components. These encodings are similar and, of course, that for dynamic resiliency subsumes that for decremental resiliency, which in turn, subsumes that for static resiliency. There is of course value in describing each of them individually because, for example, we cannot use the encoding for dynamic resiliency to focus on decremental or static resiliency only since a decrementally or statically resilient ACWF might not be dynamically resilient (so a "no" answer for dynamic would wrongly result in a "no" answer for decremental and static resiliency).

4.1 Core Encoding

Consider an arbitrary ACWF $\langle T, U, \preceq, A, C \rangle$. The core of the EGA consists of three locations: `Turn1` (initial) modeling the environment's turn, `Turn2` modeling the controller's and `Res` (accepting location) which, if reached, implies resiliency of the ACWF.

We map each user $u \in U$ into a unique incremental integer *constant* $\underline{\mathtt{u}}$ starting from 1, and we model each task $t \in T$ as an integer *variable* $\mathtt{t}$ having bounded domain $\mathtt{dom}(\mathtt{t}) = \{0, \ldots, |U|\}$, where $\mathtt{t} = 0$ means that t has not been assigned yet and $\mathtt{t} = \underline{\mathtt{u}}$ means that $\pi(t) = u$. Initially, all $\mathtt{t}$ variables are initialized to 0 (as all tasks are unexecuted). We model the availability of users as a set of variables $\{\mathtt{a}_{\underline{\mathtt{u}}} \mid u \in U\}$ each one having bounded domain $\mathtt{dom}(\mathtt{a}_{\underline{\mathtt{u}}}) = \{0, 1\}$, where $\mathtt{a}_{\underline{\mathtt{u}}} = 1$ means that u is available, whereas $\mathtt{a}_{\underline{\mathtt{u}}} = 0$ means that u is absent. Initially all these variables are set to 1 meaning that all users are available (so that the environment is free to remove anyone it likes).

Example 5. For our example, we map `alice`, `bob`, `charlie`, `david`, `emma`, `frank` and `grace` to the integer constants $1, \ldots, 7$ and in the rest of the paper we write $\underline{\mathtt{a}}$, $\underline{\mathtt{b}}$, $\underline{\mathtt{c}}$, $\underline{\mathtt{d}}$, $\underline{\mathtt{e}}$, $\underline{\mathtt{f}}$ and $\underline{\mathtt{g}}$ to refer to the integer constants representing these users. □

We also have an integer variable $\mathtt{k}$ (initialized to 0) having a bounded domain $\mathtt{dom}(\mathtt{k}) = \{0, \ldots, \underline{\mathtt{k}}\}$, where $\underline{\mathtt{k}}$ is an integer constant representing the maximum number of absent users we want to test the workflow for.

We map each counting constraint $(x, y, T_1) \in C$ into a unique incremental integer constant $\underline{\mathtt{i}}$ starting from 1.

Example 6. For our example, we have that $(1, 1, \{\mathtt{ProcR}, \mathtt{PrepC}\}) = 1$ and $(1, 1, \{\mathtt{Log}, \mathtt{PrepC}\}) = 2$ and we refer more intuitively to these constants as $\underline{\mathtt{i}}_1$ and $\underline{\mathtt{i}}_2$. □

For each counting constraint $(x, y, T_1) = \underline{\mathtt{i}}$ and user $\underline{\mathtt{u}} \in \bigcup_{t \in T_1} A(t)$, we have an integer variable $\mathtt{c}_{\underline{\mathtt{i}},\underline{\mathtt{u}}}$ having bounded domain $\mathtt{dom}(\mathtt{c}_{\underline{\mathtt{i}},\underline{\mathtt{u}}}) = \{0, \ldots, |T_1|\}$ playing the role of a counter to keep track of how many tasks $t \in T_1$ are assigned to every user u who is authorized for some of them with respect to the counting constraint (x, y, T_1). All these variables are initialized to 0 and will play a crucial role when checking the satisfaction of the corresponding counting constraint (see below).

Example 7. For our example, we have the counters $\mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{a}}}$, $\mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{b}}}$, $\mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{c}}}$, $\mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{d}}}$, $\mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{e}}}$, $\mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{b}}}$, $\mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{d}}}$, $\mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{e}}}$, where, for instance, the counter $\mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{d}}}$ is the counter for `david` with respect to $(1, 1, \{\mathtt{Log}, \mathtt{PrepC}\})$. Note that the encoding does not generate any counters for users who are not authorized for the tasks in T_1 of a counting constraint (x, y, T_1), because those users will never execute any task in T_1. □

We check that all constraints are satisfied by means of a transition `win` to move from `Turn2` to `Res` whose guard and update are

$$\langle \mathtt{Turn2}, \mathtt{OVER} \wedge \mathtt{SAT_E} \wedge \mathtt{SAT_C}, \mathtt{win}, \{\}, \mathtt{Res} \rangle$$

where the first part of the guard is

$$\mathtt{OVER} : \bigwedge_{t \in T} \mathtt{t} \neq 0$$

to check that each task has been assigned to a user. The second part of the guard is

$$\mathtt{SAT_E} : \bigwedge_{(\rho, T_1, T_2) \in C} (\bot \bigvee_{\substack{t_1 \in T_1, \\ t_2 \in T_2, \\ (u_1, u_2) \in \rho \text{ such that} \\ u_1 \in A(t_1) \wedge t_2 \in A(t_2)}} (\mathtt{t}_1 = \underline{\mathtt{u}}_1 \wedge \mathtt{t}_2 = \underline{\mathtt{u}}_2))$$

where the part "such that $u_1 \in A(t_1) \wedge t_2 \in A(t_2)$" filters ρ so that the resulting disjuncts $(\mathtt{t}_1 = \underline{\mathtt{u}}_1 \wedge \mathtt{t}_2 = \underline{\mathtt{u}}_2)$ belong to valid plans only. $\mathtt{SAT_E}$ checks that all entailment constraints are satisfied[2].

Finally, $\mathtt{SAT_C}$ does the same as $\mathtt{SAT_E}$ but with respect to counting constraints. More specifically, to check the satisfaction of a counting constraint $\underline{\mathtt{i}}$ we add several conjuncts to $\mathtt{SAT_C}$ to model conditional constraints such as "if a user u executes some task in the subset of tasks of a given $\underline{\mathtt{i}}$, then the corresponding counter $\mathtt{c}_{\underline{\mathtt{i}},\underline{\mathtt{u}}}$ must be $\geq x$ and $\leq y$. In symbols,

$$\mathtt{t} = \underline{\mathtt{u}} \implies (\mathtt{c}_{\underline{\mathtt{i}},\underline{\mathtt{u}}} \geq x \wedge \mathtt{c}_{\underline{\mathtt{i}},\underline{\mathtt{u}}} \leq y)$$

[2]The internal "$\bot$" in $\mathtt{SAT_E}$ serves to make the verification fail in case $\rho = \emptyset$ or $T_1 = \emptyset$ or $T_2 = \emptyset$.

$\mathtt{SAT_C}$ is therefore the conjunction of these implications (rewritten as disjunctions) and has the following compact form:

$$\mathtt{SAT_C}: \bigwedge_{\substack{(x,y,T_1)\in C,\\ t\in T_1, u\in A(t)}} (\mathtt{t} \neq \underline{\mathtt{u}} \vee (\mathtt{c}_{\underline{\mathtt{i}},\underline{\mathtt{u}}} \geq x \wedge \mathtt{c}_{\underline{\mathtt{i}},\underline{\mathtt{u}}} \leq y))$$

Example 8. For Figure 1 $\mathtt{SAT_E}$ and $\mathtt{SAT_C}$ are:

$$\begin{aligned}\mathtt{SAT_E}: (\bot \vee (\mathtt{ProcR} = \underline{\mathtt{b}} \wedge \mathtt{PrepC} = \underline{\mathtt{b}}) \vee (\mathtt{ProcR} = \underline{\mathtt{c}} \wedge \mathtt{PrepC} = \underline{\mathtt{b}}) \vee \\ (\mathtt{ProcR} = \underline{\mathtt{a}} \wedge \mathtt{PrepC} = \underline{\mathtt{b}}) \vee (\mathtt{ProcR} = \underline{\mathtt{a}} \wedge \mathtt{PrepC} = \underline{\mathtt{e}}) \vee (\mathtt{ProcR} = \underline{\mathtt{a}} \wedge \mathtt{PrepC} = \underline{\mathtt{d}}) \vee \\ (\mathtt{ProcR} = \underline{\mathtt{c}} \wedge \mathtt{PrepC} = \underline{\mathtt{e}}) \vee (\mathtt{ProcR} = \underline{\mathtt{b}} \wedge \mathtt{PrepC} = \underline{\mathtt{e}}) \vee (\mathtt{ProcR} = \underline{\mathtt{c}} \wedge \mathtt{PrepC} = \underline{\mathtt{d}})) \\ \wedge \\ (\bot \vee (\mathtt{ProcR} = \underline{\mathtt{a}} \wedge \mathtt{Log} = \underline{\mathtt{e}}) \vee (\mathtt{ProcR} = \underline{\mathtt{a}} \wedge \mathtt{Log} = \underline{\mathtt{d}}) \vee (\mathtt{ProcR} = \underline{\mathtt{c}} \wedge \mathtt{Log} = \underline{\mathtt{e}}) \vee \\ (\mathtt{ProcR} = \underline{\mathtt{b}} \wedge \mathtt{Log} = \underline{\mathtt{e}}) \vee (\mathtt{ProcR} = \underline{\mathtt{c}} \wedge \mathtt{Log} = \underline{\mathtt{d}})) \\ \wedge \\ (\bot \vee (\mathtt{PrepC} = \underline{\mathtt{d}} \wedge \mathtt{Sign} = \underline{\mathtt{f}}) \vee (\mathtt{PrepC} = \underline{\mathtt{e}} \wedge \mathtt{Sign} = \underline{\mathtt{g}}) \vee \\ (\mathtt{PrepC} = \underline{\mathtt{d}} \wedge \mathtt{Sign} = \underline{\mathtt{g}}) \vee (\mathtt{PrepC} = \underline{\mathtt{b}} \wedge \mathtt{Sign} = \underline{\mathtt{f}}) \vee (\mathtt{PrepC} = \underline{\mathtt{b}} \wedge \mathtt{Sign} = \underline{\mathtt{g}}))\end{aligned}$$

where the first conjunct of $\mathtt{SAT_E}$ encodes the entailment constraint $(\neg \mathit{Relatives}, \{\mathtt{ProcR}\}, \{\mathtt{PrepC}\})$, the second encodes $(\neg \mathit{Relatives}, \{\mathtt{ProcR}\}, \{\mathtt{Log}\})$ and the third encodes $(\neg \mathit{Relatives}, \{\mathtt{PrepC}\}, \{\mathtt{Sign}\})$.

$$\begin{aligned}\mathtt{SAT_C}: (\mathtt{ProcR} \neq \underline{\mathtt{a}} \vee (\mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{a}}} \geq 1 \wedge \mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{a}}} \leq 1)) \wedge (\mathtt{ProcR} \neq \underline{\mathtt{b}} \vee (\mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{b}}} \geq 1 \wedge \mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{b}}} \leq 1)) \wedge \\ (\mathtt{ProcR} \neq \underline{\mathtt{c}} \vee (\mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{c}}} \geq 1 \wedge \mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{c}}} \leq 1)) \wedge (\mathtt{PrepC} \neq \underline{\mathtt{b}} \vee (\mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{b}}} \geq 1 \wedge \mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{b}}} \leq 1)) \wedge \\ (\mathtt{PrepC} \neq \underline{\mathtt{d}} \vee (\mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{d}}} \geq 1 \wedge \mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{d}}} \leq 1)) \wedge (\mathtt{PrepC} \neq \underline{\mathtt{e}} \vee (\mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{e}}} \geq 1 \wedge \mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{e}}} \leq 1)) \\ \wedge \\ (\mathtt{Log} \neq \underline{\mathtt{d}} \vee (\mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{d}}} \geq 1 \wedge \mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{d}}} \leq 1)) \wedge (\mathtt{Log} \neq \underline{\mathtt{e}} \vee (\mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{e}}} \geq 1 \wedge \mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{e}}} \leq 1)) \wedge \\ (\mathtt{PrepC} \neq \underline{\mathtt{b}} \vee (\mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{b}}} \geq 1 \wedge \mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{b}}} \leq 1)) \wedge (\mathtt{PrepC} \neq \underline{\mathtt{d}} \vee (\mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{d}}} \geq 1 \wedge \mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{d}}} \leq 1)) \wedge \\ (\mathtt{PrepC} \neq \underline{\mathtt{e}} \vee (\mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{e}}} \geq 1 \wedge \mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{e}}} \leq 1))\end{aligned}$$

where the first conjunct of $\mathtt{SAT_C}$ encodes the counting constraint $(1, 1, \{\mathtt{ProcR}, \mathtt{PrepC}\}) = \underline{\mathtt{i}}_1$, whereas the second encodes $(1, 1, \{\mathtt{Log}, \mathtt{PrepC}\}) = \underline{\mathtt{i}}_2$.

Summing up, the core of $\mathcal{G} = \langle L, \Sigma, I, \delta, q_0, \Gamma_0, F\rangle$ of our example is:

- $L = \{\mathtt{Turn1}, \mathtt{Turn2}, \mathtt{Res}\}$.
- $I = \{\mathtt{ProcR}, \mathtt{Log}, \mathtt{PrepC}, \mathtt{Sign}, \mathtt{a}_{\underline{\mathtt{a}}}, \mathtt{a}_{\underline{\mathtt{b}}}, \mathtt{a}_{\underline{\mathtt{c}}}, \mathtt{a}_{\underline{\mathtt{d}}}, \mathtt{a}_{\underline{\mathtt{e}}}, \mathtt{a}_{\underline{\mathtt{f}}}, \mathtt{a}_{\underline{\mathtt{g}}}, \mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{a}}}, \mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{b}}}, \mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{c}}}, \mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{d}}}, \mathtt{c}_{\underline{\mathtt{i}}_1,\underline{\mathtt{e}}}, \mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{b}}}, \mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{d}}}, \mathtt{c}_{\underline{\mathtt{i}}_2,\underline{\mathtt{e}}}, \mathtt{k}\}$, where $\mathtt{dom}(t) = \{0, \underline{\mathtt{a}}, \underline{\mathtt{b}}, \underline{\mathtt{c}}, \underline{\mathtt{d}}, \underline{\mathtt{e}}, \underline{\mathtt{f}}, \underline{\mathtt{g}}\}$ for each $t \in T$, $\mathtt{dom}(\mathtt{a}_{\underline{\mathtt{u}}}) = \{0, 1\}$ for each $u \in U$, $\mathtt{dom}(\mathtt{c}_{\underline{\mathtt{i}},\underline{\mathtt{u}}}) = \{0, \ldots, |T_1|\}$ for each $(x, y, T_1) \in C$ and $u \in \bigcup_{t\in T_1} A(t_1)$, and $\mathtt{dom}(\mathtt{k}) = \{0, \ldots, \underline{\mathtt{k}}\}$.
- $\delta = \{(\mathtt{Turn2}, \mathtt{OVER} \wedge \mathtt{SAT_E} \wedge \mathtt{SAT_C}, \mathtt{win}, \{\}, \mathtt{Res})\}$.
- $\ell_0 = \mathtt{Turn1}$.

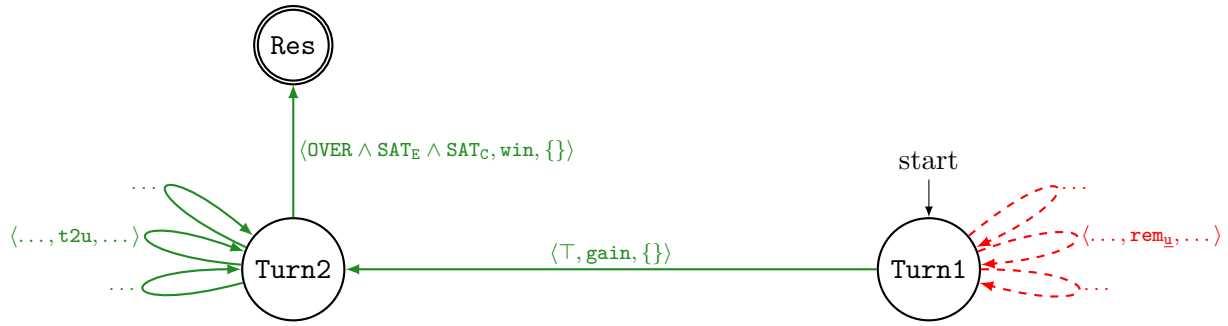


Figure 3: Skeleton of the encoding for static resiliency checking. `Turn1` is the initial location. $\texttt{rem}_{\underline{\texttt{u}}}$ transitions make users absent (environment). `t2u` transitions assign tasks to users (controller). `gain` allows the controller to play once the environment has finished making users absent, whereas `win` allows the controller to move to `Res` when all tasks have been executed and all constraints are satisfied. The environment takes uncontrollable transitions (dashed red edges), whereas the controller takes controllable ones (solid green edges).

- $\Gamma_0 = \{\texttt{ProcR} \mapsto 0, \texttt{Log} \mapsto 0, \texttt{PrepC} \mapsto 0, \texttt{Sign} \mapsto 0, \texttt{a}_{\underline{\texttt{a}}} \mapsto 1, \texttt{a}_{\underline{\texttt{b}}} \mapsto 1, \texttt{a}_{\underline{\texttt{c}}} \mapsto 1, \texttt{a}_{\underline{\texttt{d}}} \mapsto 1, \texttt{a}_{\underline{\texttt{e}}} \mapsto 1, \texttt{a}_{\underline{\texttt{f}}} \mapsto 1, \texttt{a}_{\underline{\texttt{g}}} \mapsto 1, \texttt{c}_{\underline{\texttt{i}}_1,\underline{\texttt{a}}} \mapsto 0, \texttt{c}_{\underline{\texttt{i}}_1,\underline{\texttt{b}}} \mapsto 0, \texttt{c}_{\underline{\texttt{i}}_1,\underline{\texttt{c}}} \mapsto 0, \texttt{c}_{\underline{\texttt{i}}_1,\underline{\texttt{d}}} \mapsto 0, \texttt{c}_{\underline{\texttt{i}}_1,\underline{\texttt{e}}} \mapsto 0, \texttt{c}_{\underline{\texttt{i}}_2,\underline{\texttt{b}}} \mapsto 0, \texttt{c}_{\underline{\texttt{i}}_2,\underline{\texttt{d}}} \mapsto 0, \texttt{c}_{\underline{\texttt{i}}_2,\underline{\texttt{e}}} \mapsto 0, \texttt{k} \mapsto 0\}$.
- $F = \{\texttt{Res}\}$. □

4.2 Static Resiliency

Figure 3 shows the skeleton of the encoding of an ACWF into an EGA for the static resiliency checking. The encoding adds the following transitions to the core part of the EGA discussed in Section 4.1.

To make a user $\underline{\texttt{u}}$ absent, we add an *uncontrollable* self loop transition at `Turn1` having the form

$$\langle \texttt{Turn1}, \texttt{k} < \underline{\texttt{k}} \wedge \texttt{a}_{\underline{\texttt{u}}} = 1, \texttt{rem}_{\underline{\texttt{u}}}, \{\texttt{a}_{\underline{\texttt{u}}} := 0, \texttt{k} := \texttt{k} + 1\}, \texttt{Turn1} \rangle$$

We have as many of these transitions as the number of users in U. The environment can take any of these transitions if it can still remove further users ($\texttt{k} < \underline{\texttt{k}}$) and the user it is trying to remove is still available ($\texttt{a}_{\underline{\texttt{u}}} = 1$). If this is possible, the state of the integer variable $\texttt{a}_{\underline{\texttt{u}}}$ modeling the availability of that user is flipped to 0 (meaning that u has become absent) and the current number of absent users `k` is incremented by 1.

Since uncontrollable transitions have priority over controllable ones, the environment can remove as many users as it wants (up to k) before the controller gets control of the game by taking `gain` (i.e., executing the `gain` transition) whose guard and update are:

$$\langle \texttt{Turn1}, \top, \texttt{gain}, \{\}, \texttt{Turn2} \rangle$$

The guard is always true, whereas the update makes no effect on Γ.

To assign a task `t` to a user $\underline{\texttt{u}}$, we add a *controllable* self loop transition at `Turn2` having the form

$$\langle \texttt{Turn2}, \texttt{t} = 0 \wedge \texttt{a}_{\underline{\texttt{u}}} = 1 \wedge \Pi(\texttt{t}), \texttt{t2u}, \{\texttt{t} := \underline{\texttt{u}}, \texttt{c}_{\underline{\texttt{i}}_j,\underline{\texttt{u}}} := \texttt{c}_{\underline{\texttt{i}}_j,\underline{\texttt{u}}} + 1, \dots\}, \texttt{Turn2} \rangle$$

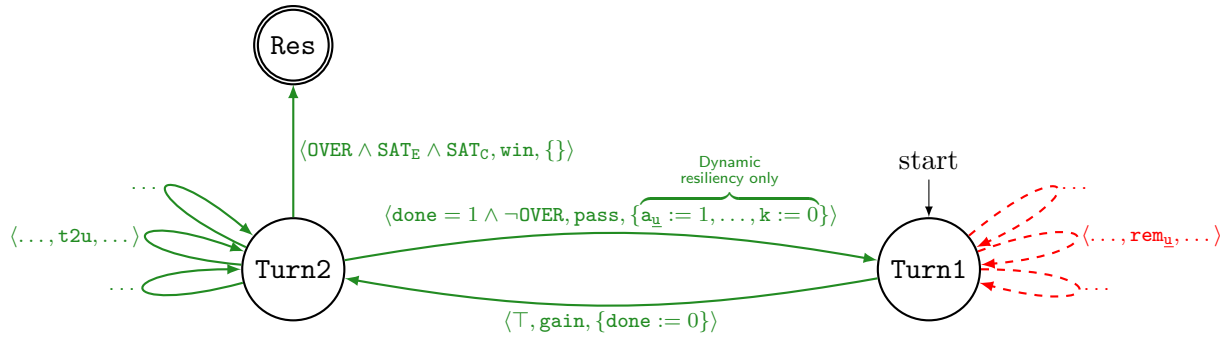


Figure 4: Skeleton of the encodings for decremental and dynamic resiliency. These encodings both add the variable `done` and the transition `pass` to regulate the two turns of the game, extend the guards and updates of `t2u` and $\texttt{rem}_{\underline{u}}$ transitions and only differ from each other for the update of the `pass` transition that in case of dynamic resiliency resets the availability state of all users.

and we have as many of them as the cardinality of the authorization relation A. The controller can assign $t \in T$ to $u \in A(t)$ if $\pi(t)$ is still undefined (modeled by $\texttt{t} = 0$) and u is available (modeled by $\texttt{a}_{\underline{u}} = 1$). Furthermore, t can be executed iff all tasks $\texttt{t}'$ occurring before it have already been executed. We model this latter condition as:

$$\Pi(\texttt{t}) : \bigwedge_{\texttt{t}' < \texttt{t}} \texttt{t}' \neq 0$$

Example 9. For `Sign2grace` we have $\Pi(\texttt{Sign}) : \texttt{Log} \neq 0 \wedge \texttt{PrepC} \neq 0$. □

The update of this transition assigns `t` to $\underline{\texttt{u}}$ and increments all counters $\texttt{c}_{\underline{\texttt{i}},\underline{\texttt{u}}}$ related to any counting constraint $(x, y, T_1) = \underline{\texttt{i}}$ such that `t` appears in T_1 and $\underline{\texttt{u}}$ is authorized for `t`.

4.3 Decremental and Dynamic Resiliency

The encodings for decremental and dynamic resiliency both extend that for static resiliency and differ from each other just for the update of a single transition. We show their skeleton(s) in Figure 4.

We add an integer variable `done` having bounded domain $\texttt{dom}(\texttt{done}) = \{0, 1\}$ and initially set to 0. We use `done` to guarantee that at any round (i) the controller assigns one and only one task to a user, and (ii) the environment waits for the controller to finish (Algorithm 2 and Algorithm 3).

We extend the `gain` transition given in Figure 3 as follows

$$\langle \texttt{Turn1}, \top, \texttt{gain}, \{\texttt{done} := 0\}, \texttt{Turn2} \rangle$$

In this way, every time the run gets to `Turn1`, the controller can execute one task only as we extend `t2u` transitions by refining the guard and update as follows

$$\begin{aligned} \langle \texttt{Turn2}, \texttt{t} = 0 \wedge \texttt{done} = 0 \wedge \texttt{a}_{\underline{\texttt{u}}} = 1 \wedge \Pi(\texttt{t}), \texttt{t2u}, \\ \{\texttt{t} := \underline{\texttt{u}}, \texttt{c}_{\underline{\texttt{i}}_j,\underline{\texttt{u}}} := \texttt{c}_{\underline{\texttt{i}}_j,\underline{\texttt{u}}} + 1, \ldots, \texttt{done} := 1\}, \texttt{Turn2} \rangle \end{aligned}$$

That is, each of these transitions can be taken only if all conditions discussed for the encoding given for static resiliency hold and furthermore the controller has not played yet in its turn (modeled by

$\texttt{done} = 0$). Once such a transition is taken, the update sets `done` to 1 (modeling "controller has played") to prevent him from taking more than one.

We add a *controllable* transition `pass` going from `Turn2` to `Turn1` to lead the run back to `Turn1` and allow the environment to remove further (or different in case of dynamic resiliency) users after the controller has finished playing. Note that the environment cannot prevent the controller from playing since `pass` requires in its guard $\texttt{done} = 1$. This transition is the only difference between the encodings for decremental and dynamic resiliency.

In case of decremental resiliency this transition is:

$$\langle \texttt{Turn2}, \texttt{done} = 1 \wedge \neg \texttt{OVER}, \texttt{pass}, \{\}, \texttt{Turn1} \rangle$$

whereas in case of dynamic resiliency it becomes:

$$\langle \texttt{Turn2}, \texttt{done} = 1 \wedge \neg \texttt{OVER}, \texttt{pass}, \{\texttt{a}_{\underline{\texttt{u}}} := 1, \ldots, \texttt{k} := 0\}, \texttt{Turn1} \rangle$$

where $\{\texttt{a}_{\underline{\texttt{u}}} := 1, \ldots\}$ is a sequence of statements operating on Γ to make all users available again and the statement $\texttt{k} := 0$ resets the counter of current absent users. $\texttt{rem}_{\underline{\texttt{u}}}$ transitions remain the same as those of the encoding given for static resiliency.

Finally, regardless of the type, static, decremental or dynamic resiliency is checked by looking for a control strategy to always eventually get to `Res`. If the workflow is resilient, a strategy for the controller exists and it is a certificate of "yes" modeling a dynamic plan π. If the workflow is not resilient, then a strategy for the environment exists and it is a certificate of "no". Such strategy allows the environment to break any execution no matter which tasks we decide to assign to which users (we will always fail).

Figure 5 shows the EGA encoding the workflow in Figure 1 for dynamic resiliency.

5 Correctness and Complexity of the Encodings

In this section, we prove that each encoding discussed in Section 4 reduces any ACWF to an EGA in polynomial time and correctly models the corresponding kind of addressed resiliency by showing that any run of the generated EGA corresponds to a run of the corresponding game defined by Wang and Li.

Before proceeding we clarify what we mean with correctness. An algorithm is *correct* if, once run on some input, it returns a correct answer for that input. An answer may be Yes/No (for decision problems) or a solution of some kind (e.g., a model for a CNF formula, an execution plan for a resilient workflow, etc.). Our approach relies on sound and complete algorithms (TCTL model checking) and corresponds to a reachability problem for extended game automata. If our approach says that a workflow is resilient, then the workflow is really so and we prove it by returning an execution plan as a "certificate of yes". If a workflow is not resilient, then our approach returns "breakable" meaning that there is no way to always satisfy the constraints during execution.

5.1 Static resiliency

Theorem 1. *The encoding for static resiliency given in Section 4.2 generates an EGA $\mathcal{G}$ in polynomial time such that the existence of a control strategy for $\mathcal{G}$ to always eventually get to* `Res` *implies static resiliency of the starting workflow.* □

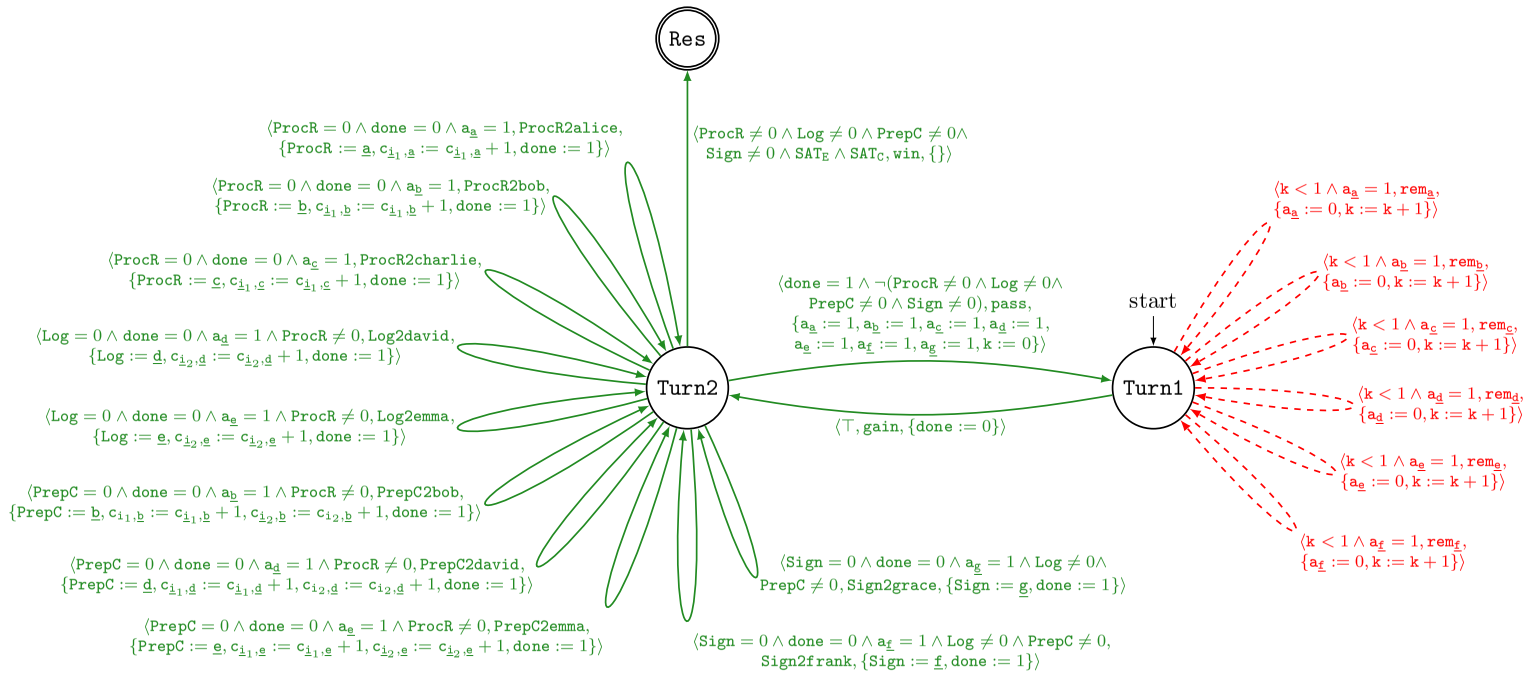


Figure 5: Extended Game Automaton encoding Figure 1. We provide details on $\texttt{SAT}_\texttt{E}$ and $\texttt{SAT}_\texttt{C}$ in the first part of Section 4.

Proof. Given a workflow $\langle T, U, \preceq, A, C\rangle$, the encoding for static resiliency given in Section 4.2 generates an EGA $\mathcal{G} = \langle L, \Sigma, I, \delta, q_0, \Gamma_0, F\rangle$ consisting of 3 locations, $|T|$ integer variables for tasks each one having a domain of $|U|+1$ elements, $|U|$ integer variables for the availability of users each one having a binary domain, $|C'| \times |U|$ integer variables for counting constraints each one having a maximum domain of $|T|+1$ elements and where $C' \subseteq C$ is the subset of counting constraints and 1 integer variable for the current number of absent users having a domain of $k+1$ elements. Then, it generates $|A|$ transitions to assign tasks to users (where each transition can have an update containing all possible counters in the worst case), $|U|$ transitions to make users absent, 1 `pass` transition and 1 `win` transition. The `win` transition contains the guard $\texttt{OVER} \wedge \texttt{SAT}_\texttt{E} \wedge \texttt{SAT}_\texttt{C}$ where `OVER` contains $|T|$ conjuncts, $\texttt{SAT}_\texttt{E}$ contains a formula whose size is $\sum_{(\rho,T_1,T_2)\in C}(1 + |\{(u_1,u_2) \mid (u_1,u_2) \in \rho \wedge t_1 \in T_1 \wedge t_2 \in T_2 \wedge u_1 \in A(t_1) \wedge u_2 \in A(t_2)\}|)$ and $\texttt{SAT}_\texttt{C}$ contains a formula whose size is $\sum_{(x,y,T_1)\in C}(\sum_{t\in T_1} |A(t)|)$. Since all these pieces are functions computable in polynomial time, the overall complexity is polynomial.

As we have already said at the end of Section 3.2, the *state* of this EGA is a pair (ℓ, Γ), where ℓ is a location and Γ the status of the integer variables.

`Turn1` is the initial location, where the environment plays. The initial state of the system is $(\texttt{Turn1}, \Gamma_0)$, where Γ_0 contains all initialized integer variables (see the end of Section 4).

When the run starts, the environment can make absent as many users as it likes (up to $\underline{\texttt{k}}$) by taking the $\texttt{rem}_{\underline{\texttt{u}}}$-transitions. Every time the environment takes one of these transitions, `k` is incremented by 1. Since all $\texttt{rem}_{\underline{\texttt{u}}}$ transitions are uncontrollable, the controller is unable to interrupt the environment by taking `gain` (the unique controllable transition having source at `Turn1`). When the environment is done, the controller can take `gain` to enter `Turn2`, the location in which it can assign tasks to (available) users. When the controller enters `Turn2`, the current state of the system is $(\texttt{Turn2}, \Gamma_i)$, where in Γ_i we have that $\texttt{k} \leq \underline{\texttt{k}}$, $\texttt{a}_{\underline{\texttt{u}}} = 0$ for each user u made absent, whereas the $\texttt{c}_{\underline{\texttt{i}},\underline{\texttt{u}}}$ and `t` variables are still the same as those in Γ_0.

This state corresponds to choosing $\texttt{Absent} \subset U$ such that $|\texttt{Absent}| \leq k$ (Algorithm 1, lines 1-2). Now, at `Turn2`, the environment will not play anymore. In this location, the controller can assign (unexecuted) tasks to (available) users by taking one and only one `t2u` transition corresponding to a pair $(t,u) \in A$. These transitions also specify conditions on predecessors (if any)[3]. When the controller is done (i.e., when all `t2u` transitions are disabled), the run can end up in two possible states.

The first possibility is the state $(\texttt{Turn2}, \Gamma'_n)$, where Γ'_n contains $\texttt{k} \leq \underline{\texttt{k}}$, $\texttt{c}_{\underline{\texttt{i}},\underline{\texttt{u}}}$ is set to a specific integer constant and $\texttt{a}_{\underline{\texttt{u}}}$ is set to 0 for each user made absent according to the specific run and $\texttt{t} = 0$ for some $t \in T$. If this is the case, it means that for some task `t` there is no authorized and available user $\underline{\texttt{u}}$ such that the partial plan extended with $\texttt{t} = \underline{\texttt{u}}$ would lead to a complete consistent one. In other words, the workflow is breakable if the environment removes all users $\underline{\texttt{u}}$ such that $\texttt{a}_{\underline{\texttt{u}}} = 0$ in Γ'_n before starting.

The second possibility is the state $(\texttt{Turn2}, \Gamma''_n)$, where Γ''_n contains $\texttt{k} \leq \underline{\texttt{k}}$, $\texttt{c}_{\underline{\texttt{i}},\underline{\texttt{u}}}$ is set to a specific integer constant and $\texttt{a}_{\underline{\texttt{u}}}$ is set to 0 for each user made absent according to the specific run and $\texttt{t} \neq 0$ for all $t \in T$. If this is the case, it means that all tasks have been assigned to a user. The environment generates such an assignment according to the precedence relation $\preceq$ and the related constraints. However, this plan is not necessarily consistent as it only verifies that (i) the plan is valid (as $(t,u) \in A$ for all $\texttt{t} = \underline{\texttt{u}}$) and (ii) all tasks are executed according to the partial

[3]This is an optimization to speed up the model checking phase pruning impossible runs. Removing this optimization would slow down the model checking phase but would not destroy the correctness of the approach.

order $\preceq$. Now, if the controller can take the win-transition, then it means that OVER, $\mathtt{SAT_E}$ and $\mathtt{SAT_C}$ valuates to true. That is, it means that a consistent plan π exists and that the workflow is statically resilient (Algorithm 1, lines 3-4). If the controller can't take the win-transition, then it means that the workflow is breakable as there is no way to satisfy the constraints with any of the possible assignments under the absence of the users removed by the environment (Algorithm 1, lines 3 and 5-6). □

Static resiliency is not a matter of which order we choose for the tasks (we just need to make sure that one exists) because users are removed before starting.

5.2 Decremental resiliency

Theorem 2. *The encoding for decremental resiliency given in Section 4.3 generates an EGA $\mathcal{G}$ in polynomial time such that the existence of a control strategy for $\mathcal{G}$ to always eventually get to* Res *implies decremental resiliency of the starting workflow.* □

Proof. For decremental resiliency the state of the EGA also contains an integer variable done that is used to regulate the interplay of the game between the controller and the environment and a pass transition to allow the environment to remove users also during execution. The complexity remains polynomial as it adds only conditions related to related done in some guards and updates.

As for static resiliency, when the run is in Turn1, the environment can take as many $\mathtt{rem_{\underline{u}}}$ transitions as it likes before starting. However, since the encoding for decremental resiliency adds pass to lead the run back to Turn1, the environment can make users absent also during execution.

Therefore, all Γs discussed for static resiliency extend by adding the variable done; for instance, in $(\mathtt{Turn1}, \Gamma_0)$ we have that Γ_0 is as for static plus $\mathtt{done} = 1$.

When the environment is done in his turn, the controller can take gain to enter Turn2.

At Turn2, before the controller starts playing, the current state of the system is $(\mathtt{Turn2}, \Gamma_i)$, where Γ_i is like Γ_0 with the difference that $\mathtt{done} = 0$ (note that gain sets done to 0 in its update). This corresponds to (extending) the set of absent users (Algorithm 2, lines 5-7).

At Turn2, the controller can make one assignment only as all t2u guards extend by adding the clause $\mathtt{done} = 0$ and all updates set done to 1. This corresponds to extending the partial plan π with one assignment only (Algorithm 2, line 8). Note that this assignment (if any) doesn't guarantee consistency. However, this is not a problem since if the run eventually gets to Res, then it must have made an assignment such that π was locally consistent. Note that the assignment of tasks to users leads to a valid plan by default since t2u transitions exist only for pairs $(t, u) \in A$. Therefore, such an assignment models Algorithm 2, lines 12-13.

The pass transition allows the controller to lead the run back to Turn1. However, this transition cannot prevent the controller from making his assignment at Turn2 because pass will be enable only when the controller is done (i.e., when it has taken a transition) and there is still some task to assign ($\mathtt{done} = 1 \wedge \neg\mathtt{OVER}$). At Turn1, the environment can again make absent a few more users (if any are left for this operation). When the environment is done with this turn (i.e., when it decides not to make any further user absent), gain allows the controller to take back control of the run by moving to Turn2 and make the next assignment. This run ends when assignments of tasks to users are no longer possible. As for static resiliency, this happens for two possibilities: $(\mathtt{Turn2}, \Gamma'_n)$ and $(\mathtt{Turn2}, \Gamma''_n)$, where $(\mathtt{Turn2}, \Gamma'_n)$ contains $\mathtt{k} \leq \underline{\mathtt{k}}$, $\mathtt{c_{\underline{i},\underline{u}}}$ is set to a specific integer constant and $\mathtt{a_{\underline{u}}}$ is set to 0 for each user made absent according to the specific run, $\mathtt{done} = 0$ and $\mathtt{t} = 0$ for some

$t \in T$, whereas $(\texttt{Turn2}, \Gamma''_n)$ contains $\texttt{k} \leq \underline{\texttt{k}}$, $\texttt{c}_{\underline{\texttt{i}},\underline{\texttt{u}}}$ is set to a specific integer constant and $\texttt{a}_{\underline{\texttt{u}}}$ is set to 0 for each user made absent according to the specific run, $\texttt{done} = 1$ and $\texttt{t} \neq 0$ for some $t \in T$.

In any of these two states the run cannot go back to `Turn1` as $\texttt{done} = 0$ (and no action of the controller sets it to 1). In the first case, the workflow is breakable, whereas in the second case, the workflow might be resilient (if the run can move to `Res`). □

It is currently unclear if decremental resiliency is a matter of order or not mainly because users may turn absent during execution, so we are not guaranteed that all total orderings for the tasks precomputed before starting are fine. What is worse is that it could not be a problem of finding the correct total order before starting, but such an order might have to be generated dynamically while executing depending on which users are absent (see below what we found for dynamic resiliency). In any case, the encoding for decremental resiliency doesn't fix any total ordering, so if an ACWF is proven decrementally resilient, the synthesized strategy will also implicitly handle this issue.

5.3 Dynamic resiliency

Theorem 3. *The encoding for dynamic resiliency given in Section 4.3 generates an EGA $\mathcal{G}$ in polynomial time such that the existence of a control strategy for $\mathcal{G}$ to always eventually get to* `Res` *implies dynamic resiliency of the starting workflow.*

Proof. The encoding for dynamic resiliency refines that for decremental resiliency by adding the restore of the availability of the users (1 statement for each user) and the reset of `k` in the update of `pass` (i.e., at the end of any turn). Therefore its complexity remains polynomial as it extends the update of `pass` of $|U| + 1$ components. Everything else remains the same. □

Dynamic resiliency *is* a matter of order. Consider the total order $\texttt{ProcR} \preceq \texttt{PrepC} \preceq \texttt{Log} \preceq \texttt{Sign}$ and assume that the controller (no matter which user is absent before starting the execution) assigns either `alice` or `charlie` to `ProcR` (and not `bob` since this assignment would only leave `emma` for `Log`). Regardless of which user the controller assigns to `ProcR`, the environment will make absent `bob` before `PrepC` is assigned. Then, since `PrepC` and `Log` must be assigned to two different users, if the controller assigns `PrepC` to `emma`, the environment will put back `bob` and make absent `david`, whereas if the controller assigns `PrepC` to `david`, then the controller will put back `bob` and remove `emma` (both before the assignment of `Log`). In this way, the only user remaining for `Log` will be the one who was assigned to `PrepC`. In such a scenario, this problem doesn't happen if `Log` executes before `PrepC`.

Consider now the total order $\texttt{ProcR} \preceq \texttt{Log} \preceq \texttt{PrepC} \preceq \texttt{Sign}$ and assume that `bob` is absent before the execution. We can assign `ProcR` to either `alice` or `charlie`. Regardless of this assignment, the environment will put back `bob` and make absent `emma` forcing the controller to assign `Log` to `david`. Then, the environment will put back `david` and make absent `bob` again forcing the controller to assign `PrepC` to `emma`. Finally, the environment will put back `bob` and make absent `grace` so that `frank` is the only user remaining for `Sign`. But since `PrepC` was assigned to `emma` and `emma` is `frank`'s daughter, no valid user has remained for `Sign`. And, of course, in this other scenario the problem doesn't occur if `PrepC` executes before `Log`.

Therefore, in dynamic resiliency the order of assigning tasks to users is definitely dynamic.

6 Erre: A Tool for Workflow Resiliency

We developed ERRE, the first tool for workflow resiliency that relies on a controller synthesis approach for the three kinds of resiliency.[4] ERRE is written in Java and acts both as a solver (by internally relying on UPPAAL-TIGA to synthesize winning strategies for EGAs) and as an executor simulator (i.e., a real time planner). Listing 1 shows ERRE's help screen.

Listing 1: ERRE's help screen.

```
Usage: java -jar erre.jar <Workflow.wf> <Action> <static|decremental|dynamic> <k> <Workflow.s> [N] [--silent]

<Action>:
  --check    internally encodes the workflow in input into an UPPAAL-TIGA specification ready to check static,
       decremental or dynamic resiliency up to k users (saves the strategy to Workflow.s)

  --execute  performs [N] (default 1) executions of the workflow in input (if resilient) according to the
      strategy (.s) synthesized by UPPAAL-TIGA.

  --silent   suppresses output (optional)

Examples:
  java -jar erre.jar CaseStudy.wf --check dynamic 2 CaseStudy.s
  java -jar erre.jar CaseStudy.wf --execute dynamic 2 CaseStudy.s 10
```

Given an ACWF $\langle T, U, \preceq, A, C \rangle$, the input language of ERRE has a well-defined context-free grammar (BNF) so as to allow yet-to-be-developed (GUI) tools to write input specifications automatically. Such a language comprises four main sections:

1. `Users` $\{\ldots\}$ specifying U.
2. `Tasks` $\{\ldots\}$ specifying T and A.
3. `Precedence` $\{\ldots\}$ specifying $\preceq$.
4. `Constraints` $\{\ldots\}$ specifying C.

As an example of ERRE's input language, Listing 2 shows the specification of the ACWF in Figure 1.

Listing 2: Specification of the ACWF in Figure 1.

```
1  # Figure 1 - The loan origination process
2  Users {
3    a b c d e f g                    # i.e., alice, bob, charlie, david, emma, frank, grace
4  }
5
6  Tasks {
7    (ProcR : a b c)                  # i.e. A(ProcR) = {alice,bob,charlie}
8    (Log : d e)                      # i.e. A(Log)  = {david,emma}
9    (PrepC : b d e)                  # i.e. A(PrepC) = {bob,david,emma}
10   (Sign : f g)                     # i.e. A(ProcR) = {frank,grace}
11 }
12
13 Precedence {
14   (ProcR <= Log)                    # i.e., ProcR is before Log
15   (ProcR <= PrepC)                  # i.e., ProcR is before PrepC
16   (Log <= Sign)                     # i.e., Log is before Sign
```

[4] The name of the tool comes from the sound of the letter "r" (for resiliency) in Italian.

```
  (PrepC <= Sign)                    # i.e., PrepC is before Sign
}

Constraints {
  # Entailment between ProC and Log (not relatives)
  ((a a) (a b) (a c) (a d) (a e) (a f) (a g) (b a) (b b) (b c) (b e) (b f) (b g) (c a) (c b) (c c) (c d) (c e)
       (c f) (c g) (d a) (d c) (d d) (d e) (d f) (d g) (e a) (e b) (e c) (e d) (e e) (e g) (f a) (f b) (f c) (
      f d) (f f) (f g) (g a) (g b) (g c) (g d) (g e) (g f) (g g) : ProcR : Log)

  # Entailment between ProC and Sign (not relatives)
  ((a a) (a b) (a c) (a d) (a e) (a f) (a g) (b a) (b b) (b c) (b e) (b f) (b g) (c a) (c b) (c c) (c d) (c e)
       (c f) (c g) (d a) (d c) (d d) (d e) (d f) (d g) (e a) (e b) (e c) (e d) (e e) (e g) (f a) (f b) (f c) (
      f d) (f f) (f g) (g a) (g b) (g c) (g d) (g e) (g f) (g g) : ProcR : PrepC)

  # Entailment between PrepC and Sign (not relatives)
  ((a a) (a b) (a c) (a d) (a e) (a f) (a g) (b a) (b b) (b c) (b e) (b f) (b g) (c a) (c b) (c c) (c d) (c e)
       (c f) (c g) (d a) (d c) (d d) (d e) (d f) (d g) (e a) (e b) (e c) (e d) (e e) (e g) (f a) (f b) (f c) (
      f d) (f f) (f g) (g a) (g b) (g c) (g d) (g e) (g f) (g g) : PrepC : Sign)

  # Counting constraint between ProcR and PrepC (separation of duty)
  (1 : 1 : ProcR PrepC)

  # Counting constraint between Log and PrepC (separation of duty)
  (1 : 1 : Log PrepC)
}
```

To check static, decremental and dynamic resiliency of the ACWF in Figure 1 (up to 1 absent user), we ran ERRE on Listing 2. We used a FreeBSD virtual machine run on top of a VMWare ESXi Hypervisor using a physical machine equipped with an Intel i7 2.80GHz and 20GB of RAM. The VM was assigned 16GB of RAM and full CPU power. ERRE proved in $\simeq$350ms that the ACWF in Figure 1 is statically resilient saving a dynamic plan of 8Kb (101-action strategy), decrementally resilient saving a dynamic plan of 8Kb (90-action strategy), and also dynamically resilient[5] saving a dynamic plan of 12Kb (141-action strategy). Details are given in Listing 3.

Listing 3: Validation of the ACWF in Figure 1 for static, decremental and dynamic resiliency.

```
$ java -jar erre.jar Fig1.wf --check static 1 Fig1.sta.s
Resiliency level = 1 (static)
k = 1
Checking Fig1.wf with UPPAAL-TIGA
Running UPPAAL-TIGA ...
Saving a 101-action strategy to Fig1.sta.s

$ java -jar erre.jar Fig1.wf --check decremental 1 Fig1.dec.s
Resiliency level = 2 (decremental)
k = 1
Checking Fig1.wf with UPPAAL-TIGA
Running UPPAAL-TIGA ...
Saving a 90-action strategy to Fig1.dec.s

$ java -jar erre.jar Fig1.wf --check dynamic 1 Fig1.dyn.s
Resiliency level = 3 (dynamic)
k = 1
Checking Fig1.wf with UPPAAL-TIGA
Running UPPAAL-TIGA ...
Saving a 141-action strategy to Fig1.dyn.s
```

[5]We invite the reader to add either `(Log <= PrepC)` or `(PrepC <= Log)` to the `Precedence` section of the specification shown in Listing 2 to reproduce the scenarios in which the workflow becomes breakable as discussed at the end of Section 4 (see below for the link to download the case study).

When an ACWF W is proved resilient up to a certaint k, ERRE saves to file (`.s`) the *dynamic plan* (i.e., the execution strategy) needed to later execute it (validate once, execute anytime). Such a plan can be used to execute the workflow for all $k' \leq k$ and all kinds of entailed resiliency (see below and Listing 5). For each kind of resiliency, we executed the ACWF in Figure 1 10000 times (Listing 4).

Listing 4: Carrying out 10000 execution simulations of the ACWF in Figure 1 with respect to static (1), decremental (2) and dynamic resiliency (3).

```
$ java -jar erre.jar Fig1.wf --execute static 1 Fig1.sta.s 10000
...
$ java -jar erre.jar Fig1.wf --execute decremental 1 Fig1.dec.s 10000
...
$ java -jar erre.jar Fig1.wf --execute dynamic 1 Fig1.dyn.s 10000
...
```

When going for the execution simulation, we can pass to ERRE an optional parameter `[N]` as the last command line argument to carry out a specific number of random simulations. If `N` $= 0$, then ERRE will only load the strategy, whereas if `N` $= 10000$ (as in Listing 4), then ERRE will load the strategy (once) and carry out 10000 execution simulations. By default, `N` $= 1$. Note that since dynamic resiliency embeds decremental and static resiliency, we can also use a dynamic strategy to carry out random simulations focusing only on decremental or static resiliency, or we can use a decremental strategy focusing only on static resiliency (Listing 5).

Listing 5: Using a single strategy for executing an ACWF with respect to different kinds of resiliency.

```
$ java -jar erre.jar Fig1.wf --execute decremental 1 Fig1.dyn.s 10000
...
$ java -jar erre.jar Fig1.wf --execute static 1 Fig1.dyn.s 10000
...
$ java -jar erre.jar Fig1.wf --execute static 1 Fig1.dec.s 10000
...
```

However, in no case we can use a static strategy for a decremental or dynamic execution (even if the ACWF is resilient for those kinds of resiliency), or a decremental strategy for a dynamic execution.

Listing 6 shows the output of a few execution simulations for static, decremental and dynamic resiliency of Figure 1 (according to the commands given in Listing 4). This example, along with ERRE and the experimental evaluation that we are about to discuss, is available at the URL `https://github.com/matteozavatteri/erre`.

Having ERRE also allowed us to carry out an experimental evaluation against a set of random ACWFs. We recall that no other tool provides exact algorithms for all three kinds of resiliency (especially dynamic) for the class of constraints we consider, so ERRE also advances the state of the art by providing experimental data for future comparisons. We generated 1000 ACWFs partitioned in 5 sets of benchmarks each one containing 100 dynamically resilient workflows (up to 2 absent users) and 100 dynamically breakable workflows (for 2 absent users). The first set `2t6u2a/` specifies workflows with 2 tasks, the second set `3t6u2a/` specifies workflows with 3 tasks and so on, up to the fifth one `6t6u2a/` that specifies workflows with 6 tasks. Regardless of the set, each workflow has exactly 6 users authorized for all tasks, at least an entailment constraint and at least a counting constraint. Furthermore, each workflow doesn't specify any partial order. Again, authorizing all users for all tasks and not restricting the partial order contributes to generating hard instances (N unordered tasks have $N!$ possible orders).

Listing 6: Execution simulations for static, decremental and dynamic resiliency. Every line in the execution starts with either `E:` meaning "environment plays" or `C:` meaning "controller plays". For static resiliency (first simulation) `frank` is absent before the execution starts. For decremental resiliency (second simulation) nobody is absent until `Log` finishes and `charlie` becomes absent after `Log`. For dynamic resiliency (third simulation) `david` is absent before starting and arrives after `ProcR` and `emma` becomes absent after `PrepC` and comes back after `Log`.

```
1  Static Resiliency               | Decremental Resiliency            | Dynamic Resiliency
2  k = 1                           | k = 1                             | k = 1
3  Execution 1                     | Execution 1                       | Execution 1
4  ------------------------------------------------------------------------------------------
5  E: Absent = {f}                 | E: Absent = {}                    | E: Absent = {d}
6  C: ProcR = a                    | C: ProcR = a                      | C: ProcR = a
7  C: Log = d                      | E: Absent = {}                    | E: Absent = {}
8  C: PrepC = b                    | C: Log = d                        | C: PrepC = b
9  C: Sign = g                     | E: Absent = {c}                   | E: Absent = {e}
10                                 | C: PrepC = b                      | C: Log = d
11                                 | E: Absent = {c}                   | E: Absent = {}
12                                 | C: Sign = f                       | C: Sign = f
13 ------------------------------------------------------------------------------------------
14 Verifying ... SAT!              | Verifying ... SAT!                | Verifying ... SAT!
```

Algorithm 4 shows the pseudo-code of the generator. In the following we discuss the data we collected.

Figure 6a shows the graphical results of the experimental evaluation in which we focused on time, where x-axes always represent the number of tasks (i.e., the set of benchmarks under analysis) and y-axes represent the average time elapsed when analyzing the instances in that set. We ran ERRE on these sets of benchmarks without imposing any timeout. For resilient workflows, we ran the analysis for static, decremental and dynamic resiliency on dynamically resilient workflows up to 2 absent users. For every color, the dashed line refers to the checking time to synthesize the strategy in memory, whereas the solid line refers to the total time to save it to disk (our test machine is not equipped with a solid state disk). For breakable workflows we run the analysis for dynamic resiliency for 2 absent users. We note that static resiliency is the easiest one to check, whereas there is not much difference between decremental and dynamic resiliency. Breaking an ACWF dynamically is easier than validating it (the analysis fails early).

Figure 6b shows the average number of actions in the specific strategies with respect to the set of benchmarks and the type of resiliency. Again, static strategies contain fewer actions, whereas decremental and dynamic ones contain of course more.

Figure 6c shows how long it takes on average to load in memory the strategies from disk of resilient workflows and then carry out 10000 random execution simulations (dashed lines). Loading time is of course the bottleneck as we are loading an exponential-size strategy from a (typically "slow") mass storage device. We can see that since the strategies are memoryless (and implemented in ERRE as hashtables "state $\rightarrow$ action") the execution is efficient (solid lines). Overall, ERRE simulated 15,000,000 of random executions. No one crashed.

Finally, Figure 6d shows the average space (on disk) consumed to save the strategies for dynamically resilient workflows.

Algorithm 4: `ACWF-Gen`(n, u, e, c)

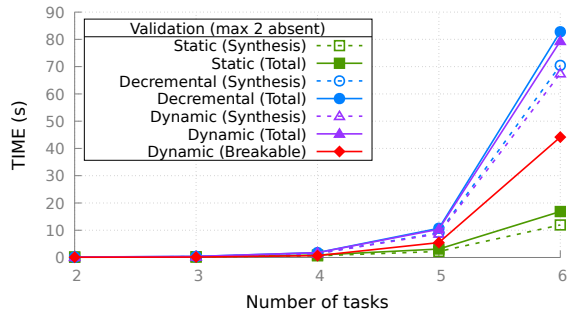
Input: An exact number of tasks n, and exact number of users u, a maximum number of entailment constraints e, a maximum number of counting constraints c.
Output: An ACWF according to Definition 1.

$W \leftarrow \langle T, U, \preceq, A, C \rangle$ ▷ Empty ACWF
▷ Generate tasks
$T \leftarrow \{t_i \mid 1 \leq i \leq t\}$
$U \leftarrow \{u_i \mid 1 \leq i \leq u\}$
▷ Generate authorization relation
for $t \in T$ **do**
  $A(t) = U$
▷ Generate entailment constraints
for $i = 1$ *to* e **do**
  $\rho \leftarrow$ a sample of $\frac{u^2}{2}$ random tuples from $U \times U$
  $T_1 \leftarrow$ a sample of random tasks from T (minimum 1)
  $T_2 \leftarrow$ a sample of random tasks from T (minimum 1)
  Generate a random Boolean b
  if $b = \top$ **then**
    $T_2 \leftarrow T_2 \setminus T_1$
  else
    $T_1 \leftarrow T_1 \setminus T_2$
  if $\rho \neq \emptyset \wedge T_1 \neq \emptyset \wedge T_2 \neq \emptyset \wedge \nexists(\rho', T_1', T_2') \in C$ *s.t.* $T_1 = T_1'$ *and* $T_2 = T_2'$ **then**
    $C \leftarrow C \cup \{(\rho, T_1, T_2)\}$
for $i = 1$ *to* c **do**
  $x \leftarrow$ random integer in $[1, 2]$
  $y \leftarrow$ random integer in $[x, x + 1]$
  $T_1 \leftarrow$ a sample of random tasks from T (min 2 and max $\frac{|T|}{2}$)
  if $\nexists(x', y', T_1') \in C$ *s.t.* $T_1 \subseteq T_1'$ *or* $T_1' \subseteq T_1$ **then**
    $C \leftarrow C \cup \{(x, y, T_1)\}$
if *no* $(\rho, T_1, T_2) \in C$ *or no* $(x, y, T_1) \in C$ **then**
  Throw away W
else
  return W

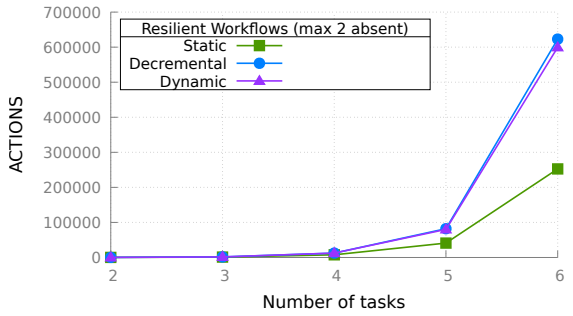
7 Related Work

In [5], Wang and Li proposed a role-and-relation based model (R^2BAC) for workflow systems and studied the complexity bounds of workflow satisfiability and resiliency. Their experimental evaluation focuses on the workflow satisfiability problem WSP, analyzing randomly generated workflow instances that differ for the number of users (1000 to 5000), or the number of tasks (10 to 40) or the number of constraints (10 to 80). Furthermore, R^2BAC does not consider counting constraints. In contrast, we gave exact algorithms to check static, decremental and dynamic resiliency and also simulate executions of resilient workflows specifying both entailment and counting constraints [22].

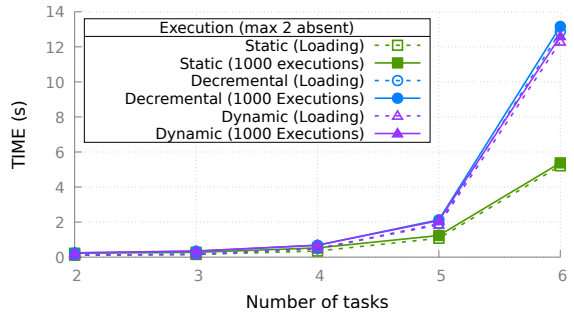
In [29], Li et al. introduced the notion of resiliency policies in the context of access control systems and defined the *resiliency checking problem (RCP)*, i.e., whether an access control state



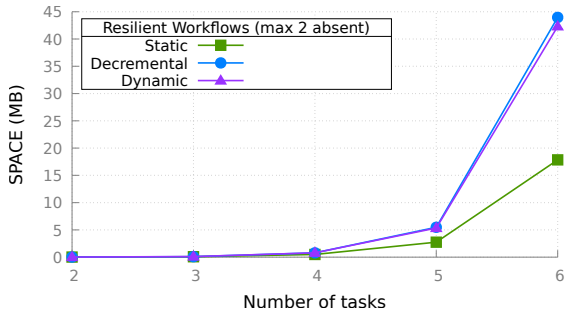
(a) *Static, decremental and dynamic resiliency checking time on all workflows.*



(b) *Average number of actions for static, decremental and dynamic strategies.*



(c) *Strategy loading time and execution on all resilient ACWFs.*



(d) *Strategy space consumption (on disk) for each kind of resiliency.*

Figure 6: Some data from the experimental evaluation with ERRE.

satisfies a given resiliency policy. They studied the complexity bound of the RCP and provided a SAT-based approach for RCP. Their experimental evaluation involves random instances with 60 to 80 users, 10 permissions and a maximum number of 3 absent users. RCP always means static resiliency, whereas we also addressed decremental and dynamic resiliency.

In [30], Crampton et al. studied the VALUED WSP to find the "least bad" plan; i.e., a plan that maximizes the number of satisfied constraints. Their experimental evaluation considers workflows with numbers of tasks ranging from 10 to 60 tasks; they analyze these workflows by comparing a pattern branch and bound (PBB) algorithm and a mixed integer programming (MIP) algorithm. Workflow resiliency is not addressed. After that, Crampton et al. [13] also proposed the BI-objective WSP as a generalization of [30]. In this case, their experimental evaluation involves workflows with numbers of tasks ranging from 10 to 36. Our work doesn't follow a "max-SAT" approach, but provides exact algorithms for static, decremental and dynamic resiliency.

In [31], Khan and Fong defined workflow feasibility (availability in some state) as the dual of workflow resiliency (availability in every state). Our work is incomparable as it doesn't tackle feasibility.

Some probabilistic approaches have been proposed to tackle resiliency. These approaches are typically faster than our approach, but, in a nutshell, that's because they may fail whereas our

approach doesn't and is exact. In particular, in [10], Mace et al. introduced the quantitative workflow resiliency as a metric of probability on how likely a workflow terminates given a security policy and a user availability model. They do so by solving a Markov Decision Process and their experimental evaluation takes into account workflows with numbers of tasks ranging from 1 to 10. Afterwards, in [11], Mace et al. provided WRAD, a tool for workflow resiliency analysis and design that encodes a workflow specification into the PRISM model checker. Workflow resiliency is defined as the maximum probability of finding a complete and valid plan. Their experimental evaluation involves the running example discussed in the paper which is a single workflow specifying 5 tasks and 3 users. Our approach cannot be used on that example because it is not probabilistic.

In [16], Paci et al. extended the RBAC-WS-BPEL language to support the specification of resiliency constraints and provided an algorithm to check if a system is failure-resistant. Although they use different terms, they deal with static resiliency only. Their experimental evaluation involves four workflows having 21 tasks and a number of users ranging from 50 to 140 (first workflow) and fixed to 50 (all other workflows). Their experimental data is not available, but, in any case, our approach does not currently handle workflows of that order of magnitude (see also the discussion on scalability in the conclusions). However, in contrast to their approach, our approach does handle decremental and dynamic resiliency.

In [15], Lowalekar et al. provided failure resiliency while satisfying security policy and constraints. They considered both static and decremental resiliency, but left the investigation for dynamic resiliency as future work. Their experimental evaluation involves randomly generated workflows with size ranging from 5 to 50 tasks and from 5 to 50 users. The authors also state that precedence relationships (with respect to the formalism they use) between tasks don't break resiliency. However, their approach doesn't involve dynamic resiliency. In contrast, we dealt with dynamic resiliency and carried out a discussion on the importance of the ordering.

In [12], Lu et al. studied the dynamic workflow adjustment, i.e., how to minimally adjust existing task-user assignments, when a sudden change, such as the absence of users, occurs. No experimental evaluation is provided. Again, we don't modify anything; we go for unbreakable security.

In [32], dos Santos et al. defined a class of *Scenario Finding Problems*, which are solutions solving the WSP that also satisfy other properties (e.g., a minimal number of users must be present). Their experimental evaluation involves two real world examples whose size are 7 tasks and 2 SoD constraints, and 9 tasks and 3 SoD constraints, respectively. Afterwards, in [14], dos Santos et al. solved static workflow resiliency by pre-computing reachability graphs by model checking the system, but they didn't address decremental resiliency nor dynamic resiliency. The experimental evaluation involves the same two workflows considered in [32]. Again, while we do not provide a comparison on these two examples for static resiliency as it would not be particularly meaningful, we wish to remark that our approach also addresses decremental and dynamic resiliency.

In this paper we dealt with (dynamic) controllability analysis to answer the workflow resiliency problem. Although focusing on slightly different areas, several papers studying dynamic controllability with or without employing (extended timed) game automata helped to devise our approach (see for example, [**?**, 33, 34, 35, 36, 37] for an example of dynamic controllability of temporal networks).

In [38], Combi et al. provided a language to model temporal plans and introduced security constraints as a means to prevent users from executing tasks if a temporal constraint is violated. Algorithms for satisfiability and resiliency are left for future work. After that, in [33], Combi et

al. proposed *Access Controlled Temporal Networks* to model temporal plans (under temporal and conditional uncertainty) augmented with users and authorization constraints. The assignment of the users depends on how the uncontrollable parts behave ("dynamic WSP"). However, workflow resiliency is left as future work. Instead, focusing on resource controllability only, in [6], Zavatteri et al., proposed an initial approach to check weak, strong and dynamic controllability for access controlled workflows under conditional uncertainty by mapping workflow paths to constraint networks (CNs) [7] and reasoning on the intersection of common parts. The proposed approach pointed out that dynamic controllability might be a matter of how the components of the workflow are ordered, an hypothesis that was later confirmed by Zavatteri e Viganò in [8, 9] with the proposal of constraint networks under conditional uncertainty (CNCUs). The experimental evaluations of CNCUs can be seen as evaluations for a conditional WSP (i.e., a WSP with uncontrollable XOR-splits considered). Having made such an assumption, the size of these conditional workflows provided in [8] ranges from 5 to 15 tasks (where 1 to 10 are uncontrollable XOR splits) and 10 to 30 authorized users for each task, whereas the workflows provided in [9] have a fixed number of 6 tasks with the same 6 users authorized for each task, no partial order and a number of XOR-splits ranging from 1 to 6 (this shows again that even small instances can be hard). In this paper we have seen that workflow resiliency shares with these latter works the matter of order. However, all these last works don't deal with the absence of resources.

8 Conclusions and Future Work

We started from the definitions (and corresponding games) provided by Wang and Li in [5, 4] for static, decremental and dynamic workflow resiliency and we considered the class of constraints defined in [22] (in [5, 4] counting constraints are not employed). We provided three encodings into extended game automata to model these games, and we proved that our encodings are correct and are generated in polynomial time. ACWFs that are resilient are (dynamically) satisfiable, the vice versa does not hold in general. We employed Uppaal-TIGA as an off the shelf model checker for solving these two-player reachability games. If the ACWF is resilient, i.e., if a winning strategy for the controller exists, Uppaal-TIGA returns in output such a strategy. If the ACWF is breakable, Uppaal-TIGA returns a *counter-strategy* for the environment allowing him to always *break* the execution (i.e., prevent the controller from entering `Res`).

To automate the whole process, we developed Erre, the first tool for workflow resiliency relying on controller synthesis. Erre allows for an automated model generation encoding into an EGA a specification of an ACWF taken as input. Then, Erre internally relies on Uppaal-TIGA to prove that the workflow is either resilient or breakable. If the ACWF is resilient, Erre compresses (online) the strategy returned in output by Uppaal-TIGA and saves it to file. Finally, Erre also allows one to carry out random execution simulations as (further) evidence that everything discussed in this paper works correctly. On the one hand, Uppaal-TIGA guarantees that the algorithms employed to answer the *decision* problem of workflow resiliency are sound and complete, on the other hand Erre guarantees that the approach is fully automated from analysis to simulation. In this way, we provided a *usable* approach even for designers with little or no knowledge on EGAs. With Erre, we carried out a preliminary experimental evaluation against a set of benchmarks we generated in order to compare time performances and space consumption of the three main kinds of resiliency. We also made available online all the results of the experiments for the world to compare. Scalability is a very interesting problem. We provided data to show that even small

workflow instances can have characteristics that result in the model checking phase to blow up. In our experimental evaluation, each workflow instance has 6 tasks and each task has the same 6 authorized users. Furthermore, no partial order is imposed. These characteristics allow the model checking phase to explore a huge number of states. We could have provided workflows with more tasks, imposing a total order and saying that the approach is (a bit more) scalable, but this would not have been fully true and thus we decided not to. There are complexity bottlenecks in these problems (more than a tool bottleneck). When any kind of resiliency is checked by setting $k = 0$, all three encodings boil down to check classic workflow satisfiability (in that case, the encoding for static is the best to use as it does not have any "game interplay" during execution).

Static resiliency is not a matter of order, whereas dynamic is so. It remains unclear if decremental resiliency is affected by that issue. Therefore, in addition to looking for further optimizations, future work will target possible ordering problems in decremental resiliency. Another direction worth following is that of investigating if static resiliency remains coNP^{NP}-complete and if decremental and dynamic resiliency remain PSPACE-complete also with respect to the class of constraints described in [22].

Our approach is relevant not just for the communities working on business processes and workflows and, more widely, artificial intelligence, but also for the security community. So, let us now take stock of our results and clarify their significance for security. As we have already remarked, ACWFs augment classic workflows by adding users and authorization constraints. Users execute tasks, whereas authorization constraints specify which users are (still) authorized to execute the remaining tasks depending on who did what. Therefore, authorization constraints are primitives for the specification of security policies. Checking workflow satisfiability ensures that an assignment of users to tasks satisfying all authorization constraints exists. Workflow satisfiability does not consider the absence of users, therefore no execution of the ACWF breaks security (i.e., the authorization constraints) if a consistent plan exists and it is followed.

However, when it comes to absent users, we need resiliency analysis to keep guaranteeing unbreakable security. The approach in this paper validates ACWFs against the three well-known types of resiliency defined by Wang and Li. As a result, our approach guarantees that no execution breaks security if a dynamic plan exists and it is followed. If it broke them, our analysis would find it out as in the model checking phase, we would be able to synthesize a counter-strategy for the environment to break any execution. In this context, this counter-strategy can be seen as an attack to the correct execution of the ACWF[6] and/or to the security policies the workflow is supposed to enforce. If no counter-strategy exists, it means that there exists a dynamic plan for the controller able to react to *any* possible move of the environment always telling the controller the right move to make in order to get to the end satisfying all authorization constraints (i.e., without breaking security). Therefore, resilient workflows guarantee unbreakable security.

References

[1] C. Combi, M. Gambini, S. Migliorini and R. Posenato, Representing Business Processes Through a Temporal Data-Centric Workflow Modeling Language: An Application to the

[6] There might be times in which the environment breaks the execution by making absent all users authorized for a task. If this happens, security policies might still hold (i.e., the partial plan could be locally consistent), but the execution will never get to the end.

Management of Clinical Pathways, *IEEE Transactions On Systems, Man, and Cybernetics: Systems* **44**(9) (2014), 1182–1203.

[2] R. Lenz and M. Reichert, IT support for healthcare processes - premises, challenges, perspectives, *Data & Knowledge Engineering* **61**(1) (2007), 39–58.

[3] H.A. Reijers and J. Mendling, Modularity in Process Models: Review and Effects, in: *6th International Conference on Business Process Management (BPM 2008)*, Lecture Notes in Computer Science, Vol. 5240, Springer, 2008, pp. 20–35.

[4] Q. Wang and N. Li, Satisfiability and Resiliency in Workflow Systems, in: *12th European Symposium On Research In Computer Security (ESORICS 2007)*, Lecture Notes in Computer Science, Vol. 4734, Springer, 2007, pp. 90–105.

[5] Q. Wang and N. Li, Satisfiability and Resiliency in Workflow Authorization Systems, *Transactions on Information and System Security (TISSEC)* **13**(4) (2010).

[6] M. Zavatteri, C. Combi, R. Posenato and L. Viganò, Weak, Strong and Dynamic Controllability of Access-Controlled Workflows Under Conditional Uncertainty, in: *15th International Conference on Business Process Management (BPM 2017)*, Lecture Notes in Computer Science, Vol. 10445, Springer, 2017, pp. 235–251. doi:10.1007/978-3-319-65000-5_14.

[7] R. Dechter, *Constraint processing*, Elsevier, 2003.

[8] M. Zavatteri and L. Viganò, Constraint Networks Under Conditional Uncertainty, in: *10th International Conference on Agents and Artificial Intelligence (ICAART 2018)*, SciTePress, 2018, pp. 41–52. doi:10.5220/0006553400410052.

[9] M. Zavatteri and L. Viganò, Conditional Uncertainty in Constraint Networks, in: *Agents and Artificial Intelligence*, Lecture Notes in Computer Science, Vol. 11352, Springer, 2019, pp. 130–160. doi:10.1007/978-3-030-05453-3_7.

[10] J.C. Mace, C. Morisset and A. van Moorsel, Quantitative Workflow Resiliency, in: *19th European Symposium on Research in Computer Security (ESORICS 2014)*, Lecture Notes in Computer Science, Vol. 8712, Springer, 2014, pp. 344–361.

[11] J.C. Mace, C. Morisset and A. van Moorsel, WRAD: Tool Support for Workflow Resiliency Analysis and Design, in: *Software Engineering for Resilient Systems: 8th International Workshop (SERENE 2016)*, Lecture Notes in Computer Science, Vol. 9823, Springer, 2016, pp. 79–87.

[12] H. Lu, Y. Hong, Y. Yang, Y. Fang and L. Duan, Dynamic Workflow Adjustment with Security Constraints, in: *28th IFIP Conference on Data and Applications Security and Privacy (DBSec 2014)*, Lecture Notes in Computer Science, Vol. 8566, Springer, 2014, pp. 211–226.

[13] J. Crampton, G. Gutin, D. Karapetyan and R. Watrigant, The bi-objective workflow satisfiability problem and workflow resiliency, *Journal of Computer Security* **25**(1) (2017), 83–115.

[14] D.R. dos Santos, S. Ranise, L. Compagna and S.E. Ponta, Automatically finding execution scenarios to deploy security-sensitive workflows, *Journal of Computer Security* **25**(3) (2017), 255–282.

[15] M. Lowalekar, R.K. Tiwari and K. Karlapalem, Security Policy Satisfiability and Failure Resilience in Workflows, in: *The Future of Identity in the Information Society: FIDIS International Summer School Revised Selected Papers*, Springer, 2009, pp. 197–210.

[16] F. Paci, R. Ferrini, Y. Sun and E. Bertino, Authorization and User Failure Resiliency for WS-BPEL Business Processes, in: *6th International Conference on Service-Oriented Computing (ICSOC 2008)*, Lecture Notes in Computer Science, Vol. 5364, Springer, 2008, pp. 116–131.

[17] D.R. dos Santos and S. Ranise, A Survey on Workflow Satisfiability, Resiliency, and Related Problems. `http://arxiv.org/abs/1706.07205` (2017).

[18] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

[19] O. Maler, A. Pnueli and J. Sifakis, On the synthesis of discrete controllers for timed systems, in: *12th Annual Symposium on Theoretical Aspects of Computer Science Munich (STACS 1995)*, Lecture Notes in Computer Science, Vol. 900, Springer, 1995, pp. 229–242.

[20] G. Behrmann, A. David and K.G. Larsen, A Tutorial on Uppaal, in: *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, (SFM-RT 2004), Revised Lectures*, Lecture Notes in Computer Science, Vol. 3185, Springer, 2004, pp. 200–236.

[21] K.G. Larsen, P. Pettersson and W. Yi, UPPAAL in a Nutshell, *Software Tools for Technology Transfer (STTT)* **1**(1–2) (1997), 134–152.

[22] J. Crampton, G. Gutin and A. Yeo, On the Parameterized Complexity and Kernelization of the Workflow Satisfiability Problem, *Transactions on Information and System Security (TISSEC)* **16**(1) (2013).

[23] P.W.L. Fong, Results in Workflow Resiliency: Complexity, New Formulation, and ASP Encoding, in: *9th ACM Conference on Data and Application Security and Privacy (CODASPY'19)*, ACM, 2019, pp. 185–196.

[24] E. Asarin, O. Maler and A. Pnueli, Symbolic Controller Synthesis for Discrete and Timed Systems, in: *International Hybrid Systems Workshop (Hybrid Systems II)*, Lecture Notes in Computer Science, Vol. 999, Springer, 1995, pp. 1–20.

[25] G. Behrmann, A. Cougnard, A. David, E. Fleury, K.G. Larsen and D. Lime, UPPAAL-Tiga: Time for Playing Games!, in: *International Conference on Computer Aided Verification (CAV 2007)*, Lecture Notes in Computer Science, Vol. 4590, Springer, 2007, pp. 121–125.

[26] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson and W. Yi, UPPAAL – a Tool Suite for Automatic Verification of Real-time Systems, in: *International Hybrid Systems Workshop (Hybrid Systems III): Verification and Control*, Lecture Notes in Computer Science, Vol. 1066, Springer, 1996, pp. 232–243.

[27] E.M. Clarke, E.A. Emerson and A.P. Sistla, Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, *Transactions on Information and System Security (TISSEC)* **8**(2) (1986), 244–263.

[28] R. Alur, C. Courcoubetis and D. Dill, Model-Checking in Dense Real-Time, *Information and Computation* **104**(1) (1993), 2–34.

[29] N. Li, Q. Wang and M. Tripunitara, Resiliency Policies in Access Control, *Transactions on Information and System Security (TISSEC)* **12**(4) (2009), 20–12034.

[30] J. Crampton, G. Gutin and D. Karapetyan, Valued Workflow Satisfiability Problem, in: *20th ACM Symposium on Access Control Models and Technologies (SACMAT 2015)*, ACM, 2015, pp. 3–13.

[31] A.A. Khan and P.W.L. Fong, Satisfiability and Feasibility in a Relationship-Based Workflow Authorization Model, in: *17th European Symposium on Research in Computer Security (ESORICS 2012)*, Lecture Notes in Computer Science, Vol. 7459, Springer, 2012, pp. 109–126.

[32] D.R. dos Santos, S. Ranise, L. Compagna and S.E. Ponta, Assisting the Deployment of Security-Sensitive Workflows by Finding Execution Scenarios, in: *29th IFIP Conference on Data and Applications Security and Privacy (DBSec 2015)*, Lecture Notes in Computer Science, Vol. 9149, Springer, 2015, pp. 85–100.

[33] C. Combi, R. Posenato, L. Viganò and M. Zavatteri, Access Controlled Temporal Networks, in: *9th International Conference on Agents and Artificial Intelligence (ICAART 2017)*, ScitePress, 2017, pp. 118–131. doi:10.5220/0006185701180131.

[34] M. Cairo, C. Combi, C. Comin, L. Hunsberger, R. Posenato, R. Rizzi and M. Zavatteri, Incorporating Decision Nodes into Conditional Simple Temporal Networks, in: *24th International Symposium on Temporal Representation and Reasoning (TIME 2017)*, LIPIcs, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.TIME.2017.9.

[35] M. Zavatteri, Conditional Simple Temporal Networks with Uncertainty and Decisions, in: *24th International Symposium on Temporal Representation and Reasoning (TIME 2017)*, LIPIcs, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.TIME.2017.23.

[36] M. Zavatteri and L. Viganò, Conditional simple temporal networks with uncertainty and decisions, *Theoretical Computer Science* (available online 2018). doi:10.1016/j.tcs.2018.09.023.

[37] C. Combi, R. Posenato, L. Viganò and M. Zavatteri, Conditional Simple Temporal Networks with Uncertainty and Resources, *Journal of Artificial Intelligence Research (to appear)* (2019).

[38] C. Combi, L. Viganò and M. Zavatteri, Security Constraints in Temporal Role-Based Access-Controlled Workflows, in: *6th ACM Conference on Data and Application Security and Privacy (CODASPY 2016)*, ACM, 2016, pp. 207–218. doi:10.1145/2857705.2857716.